José Maria Pantoja Mata Vale e Azevedo

Licenciado em Engenharia Informática

Image Stream Similarity Search in GPU Clusters

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Hervé Paulino, Assistant Professor,
Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa

Co-orientador: João Magalhães, Assistant Professor,
Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa

Júri

Presidente: Doutor João M. S. Lourenço
Arguentes: Doutor João Miguel Duarte Ascenso
Doutor Hervé Miguel Cordeiro Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2018

Image Stream Similarity Search in GPU Clusters

Copyright © José Maria Pantoja Mata Vale e Azevedo, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

À memória do meu avô Neka.

Acknowledgements

I am grateful for my advisers, Prof. Doutor Hervé Paulino and Prof. Doutor João Magalhães for their continued support throughout the conception, research and elaboration of this thesis. I am thankful for the great institution that is FCT/UNL, its professors, employees and fellow students. They provided me with the best education I could hope for, and granted me tools that I am sure will use throughout my whole life. I could not be more happy that I frequented this institution, and these are five years I am sure will never forget.

I would like to thank my father, Álvaro Vale e Azevedo, who constantly inspires and enables me to become the best engineer possible, following his footsteps. I would also like to thank my mother, Joana Pantoja Mata, because without her insistence and concern for my future, I wouldn't be half the student I am. I could not be more grateful for the parents I have, who worked hard their whole life, and sacrificed time of their own to see me achieve success.

Finally, I have to thank all of my friends. Each and every one of them has, in some way, contributed to making my life happier, and they do so on a daily basis. They helped me in the countless sleepless nights I had throughout this course, and they helped me to relax and wind down after every test, exam and project. After every week, or month of stress they are exactly who I want to see. They fight and cheer for my success as much as I fight and cheer for theirs.

Abstract

Images are an important part of today's society. They are everywhere on the internet and computing, from news articles to diverse areas such as medicine, autonomous vehicles and social media. This enormous amount of images requires massive amounts of processing power to process, upload, download and search for images. The ability to search an image, and find similar images in a library of millions of others empowers users with great advantages. Different fields have different constraints, but all benefit from the quick processing that can be achieved.

Problems arise when creating a solution for this. The similarity calculation between several images, performing thousands of comparisons every second, is a challenge. The results of such computations are very large, and pose a challenge when attempting to process. Solutions for these problems often take advantage of graphs in order to index images and their similarity. The graph can then be used for the querying process. Creating and processing such a graph in an acceptable time frame poses yet another challenge.

In order to tackle these challenges, we take advantage of a cluster of machines equipped with Graphics Processing Units (GPUs), enabling us to parallelize the process of describing an image visually and finding other images similar to it in an acceptable time frame. GPUs are incredibly efficient at processing data such as images and graphs, through algorithms that are heavily parallelizable. We propose a scalable and modular system that takes advantage of GPUs, distributed computing and fine-grained parallellism to detect image features, index images in a graph and allow users to search for similar images.

The solution we propose is able to compare up to 5000 images every second. It is also able to query a graph with thousands of nodes and millions of edges in a matter of milliseconds, achieving a very efficient query speed. The modularity of our solution allows the interchangeability of algorithms and different steps in the solution, which provides great adaptability to any needs.

Keywords: computer vision; distributed computing; graphics processing unit; stream processing; computer systems and networks; high-performance computing;

Resumo

As imagens são uma parte vital da sociedade actual. Existem em todo o lado na internet e na computação, desde artigos de notícias a áreas tão diversas como a medicina, veículos autónomos e redes sociais. Esta quantidade enorme de imagens necessita de um poder de processamento suficiente para as processar. A possibilidade de pesquisar uma imagem e encontrar imagens semelhantes a qualquer outra, permite a utilizadores de variadas áreas navegar as grandes quantidades de dados que existem, de uma forma rápida e fácil. Algumas áreas têm requisitos diferentes, contudo todas beneficiam do rápido processamento que pode ser alcançado.

Diversos problemas ocorrem ao criar uma solução para estes desafios. O cálculo de semelhança entre diversas imagens, e a execução de milhares de comparações a cada segundo, é um desafio. Os resultados de computações como estas são extremamente grandes, e criam um desafio de processamento. Soluções para estes problemas tipicamente usam grafos com o intuito de indexar imagens e a sua semelhança. O grafo pode então ser usado para o processo de pesquisa. A criação e processamento de um grafo deste tipo apresenta ainda outro desafio.

De forma a solucionar estes desafios, tiramos partido de um *cluster* de máquinas equipadas com Unidades de Processamento Gráficos (do inglês Graphics Processing Unit, GPU), que nos permite paralelizar o processo de descrever uma imagem visualmente e encontrar outras imagens semelhantes numa janela de tempo aceitável. Os GPUs são extremamente eficientes a processar dados tais como imagens e grafos, através de algoritmos extremamente paralelizáveis. Propomos um sistema escalável e modular que, através dos benefícios dos GPUs, da computação distribuída e do paralelismo de grão fino, detecta características visuais em imagens, indexa-as num grafo e permite que os utilizadores procurem por imagens semelhantes.

A solução que propomos tem a capacidade de comparar até 5000 imagens a cada segundo. Consegue alcançar também o rápido processamento de pesquisas num grafo com milhares de nós e milhões de arcos, numa questão de milisegundos. A modularidade da nossa solução permite a fácil mudança de algoritmos e passos diferentes da mesma, o que permite uma grande adaptação a qualquer necessidade.

Palavras-chave: visão computacional; computação distribuída; unidade de processamento de gráficos; processamento de *streams*; redes e sistemas de computadores; computação de alto desempenho;

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Images and videos are everywhere in the modern days. They are an integral part of our media and influence the way we communicate with each other. Certain events are captured and immortalized through images, and shared massively on websites like Twitter, Facebook and Instagram. Nowadays, we are able to live an event such as a popular sporting event or a catastrophe through the eyes of hundreds of thousands to millions of people through the images they share on social media. There is significant power in being able to process the massive amounts of images shared on platforms like these, a task no human can achieve.

Being able to search for an image of a certain object, event or person in a library of millions of images enables a user to have a different perspective on that event and is a powerful ally for journalists and law enforcement authorities. Also, the ability of organizing different images of an event (*e.g.* the football World Cup) by the different days and locations where the pictures take place enable users to view the event in a much simpler and organized fashion. Traffic and surveillance cameras also require the processing of images in order to identify faces and pedestrians. Fields such as medical research and diagnostics benefit greatly from comparing the image of a patient's exam against a huge library of other exams, enabling the quick identification of patients with similar conditions. Emerging technologies such as autonomous vehicles also require quick image processing in order to identify roads, obstacles, pedestrians, traffic signs, crosswalks and other vehicles. They even require identifying possible obstacles that have not been seen before. All of these applications (along with many others) have different types of constraints, but they all require processing potentially billions of images, some even continuously, as more images are created (possibly in the thousands every second).

We require severe computational power, as well as specialized algorithms in order to keep up with such demands. Graphics processing units (GPUs) are a prime candidate for processing these types of computations, as they are able to achieve massive parallelism in algorithms such as the ones required for these types of problems. Harnessing all of the computational power available in the current days is also a significant task. We propose to combine the power of GPUs and distributed computing in order to tackle these problems and achieve sufficient computational power for fast image processing and searching, by developing a scalable system that takes advantage of the computational power of a cluster both vertically and horizontally. A library like this is very useful in modern days, allowing programmers to focus more on their task rather than in the harnessing of computational power, as well as worrying about the intricacies of distributed computing.

1.2 Problem

The main problem we need to tackle is the similarity calculation between potentially millions of images, performing thousands of image comparisons every second and considering these images may be consumed from a stream. This means processing the massive amounts of images in an acceptable time frame and having the ability of processing image streams containing hundreds to thousands of images every second. In order to tell images from each other, we require the extraction of image features. Each image is described by several features, and features from different images can be compared in order to determine the similarity between them. Nowadays, there are several algorithms that have the ability of computing features from images, each with different levels of speed, precision and efficiency. This step is computationally intensive and benefits greatly from parallelization. We can then use such features for yet another computation: image similarity. By comparing image features against each other, we are able to obtain a similarity score, which tells us how similar two images are.

In order to be able to query a huge library of images for similar images, a certain query image needs to be compared against all of the images present in the library, for more precise results. Its also possible to use search-space reduction techniques to reduce the number of comparisons that need to be made, although we do not tackle this problem in this thesis.

If we are to compare images to all the images already present in the library, the computational work will escalate exponentially. This means that this is yet another step that benefits enormously from (and, arguably, requires) parallelism, and efficient use of computing resources. Images need to be related to each other, and a typical approach in these types of problems is the usage of graphs. Images can be organized as nodes in a graph, where the edges dictate the similarity between each node. This facilitates the query process, by being able to process and search the graph. The construction and processing of this graph is yet another challenge due to the distribution of this process through several machines and the memory constraints of each machine. A graph of this

sort, harbouring thousands of nodes and millions of edges is difficult to construct and manage. If the graph is big enough, it may not fit into memory. The graph may need to be partitioned and split through several machines, which raises the problem of how to address queries in a distributed graph.

There is an obvious advantage in distributing the whole process through several machines and using GPUs for computation, but this distribution introduces several problems that need to be addressed, such as workload balancing and partitioning. If work is not correctly balanced through machines, the distribution advantage becomes smaller and smaller, or worse off, we may obtain incorrect results without realising it (*i.e.* race conditions). The datasets need to be correctly partitioned and distributed in order to avoid redundant computations, as well as correctly aggregated in order to avoid losing computation. Yet more problems arise when querying a certain image for similarity, considering the whole library of images may be distributed through several machines. Which machines contain the similar images we wish to find? How do we find them? Can we compare an image against the whole dataset in an acceptable time frame? How can we deal with race conditions? Can we take advantage of fine-grained parallelism to make this process more efficient? It is important to carefully consider which parts of our solution require more parallelism, which computations we can sacrifice in benefit of others, in order to be as efficient as possible. These are all issues that need to be taken into consideration and solved if we are to take advantage of the power of distributed computing for this problem.

This thesis is the first step in the building of a system that tackles these problems. The focus of the thesis is to build a functional system that is able to process image streams and compare thousands of images per second, outputting graphs that are used to perform queries and retrieve image similarity. We consider that each machine possesses the whole image graph, and it is not distributed through several machines. This simplifies the query process but introduces possible memory constraints (regarding the amount of images that can be processed simultaneously). We also perform comparisons for every pair of images that the stream provides, a computationally intensive task, but still a precise one. As mentioned above, the search-space reduction (reduction of image comparisons) problem is not addressed in this work.

1.3 Solution

As hinted at before, the solution passes through using the computational power of a distributed cluster of machines equipped with GPUs. Most computations that our solution requires are highly parallelizable operations, meaning that GPUs can achieve significant performance at a much higher level than would be achievable on a CPU. Besides the use of GPUs, the usage of several machines enables us to distribute the workload by assigning different machines to compute different parts of our solution. In summary, our solution can be broken down in the following components:

- Image download
- Image preprocessing
- Image feature detection and extraction
- Feature matching
- Similarity Computation
- Graph construction

These steps consist of the computations our solution must execute in order to build a graph containing a library of images that we can later search. This graph can also be iterated several times, by adding more images to our pipeline in order to be processed, further increasing the diversity of the graph. After the graph is built, we will then be able to search it through the following order of operations:

- Query image graph indexation
- Graph search operation

The first step consists in the execution of the first pipeline listed, but only for the query image. Once it is indexed in the graph along will all the previous images that were processed, we can execute the last step, which consists in the execution of an algorithm in order to find the closest images to the query image. In the context of an application, if we were to take the image of a building, for instance, we would be able to observe other pictures of that same building (perhaps in different perspectives, times of the day or seasons) or pictures of similar buildings (with similar architecture and facade, for instance). This, of course, assumes our library contains said images. This means a richer, more complete library will have a chance of yielding better results for image similarity search, but will also take longer to search through and be harder to maintain and increment.

Our library also plans to take advantage of image metadata in order to dictate when in time the picture was taken. Using this timestamp, we can compare images by also taking time into account. We can factor it in the similarity computation (by combining the visual similarity with the time difference) or simply query how close in time any two images are.

1.3.1 Execution Model

The execution model for our library can be observed in figure 1.1. The green rectangle consists of the hardware that will execute our solution, while the red rectangles detail the pipeline each machine executes. m machines are tasked with processing the image stream, whichever it may be, and retrieving the images (*i.e.* downloading them) and extracting features from the images, using one or more feature extraction algorithms (as detailed in

Chapter 2). This process takes advantage of the GPU, to speed up the feature extraction process. Image features are then forwarded to n machines, which execute the feature matching process, calculating a similarity score between incoming images. These scores are finally indexed in a graph, to be processed later.

The reason we have more machines executing feature matching and graph processing than machines executing feature extraction is due to the fact that this is a much slower, more computationally intensive process, when compared to image downloading and feature extraction. Its important to separate these steps and find a balance in the number of machines that execute them, so that performance is maximized. The need for this separation (and to have more machines execute feature matching than feature extraction) is evidenced by the fact that every image must be compared against every other image, a process that becomes slower and slower as more images arrive. Also, this way, since only image features are shared between machines (instead of actual images), we reduce strain on the network from communication betweens machines, that could be introduced by the constant communication of large images (tens to hundreds of *megabytes*) between machines. Feature vectors are typically much smaller in size and constitute a very reliable representation of an image.

The feature detection and extraction point serves to compute image descriptors (*i.e.* feature vectors), enabling future comparison. The feature matching section takes these image descriptors and uses them to compare images against each other, outputting a value (or score) which represents how similar the two images are. The final step constructs or updates the image similarity graph, which can then be used for several types of queries. These steps and their implementations are further detailed in Chapter 2 and Chapter 3. Most of the steps are parallelized and executed on the GPU, which maximizes performance.

As can be seen in Figure 1.1, the key idea is to have m machines execute feature extraction (using one or more algorithms), while n other machines feed on that information in order to compute the graph, a process that is continuous and iterative. Typically $m < n$ because, as we explained previously, the feature matching and graph construction step is more computationally expensive than the image processing and feature extraction steps.

Taking a close look at Figure 1.1, the first m machines process the stream (or streams), download the images and extract features from them (step 1). These machines can even extract features from multiple computer vision algorithms (which have different use cases, further detailed in Chapter 2). This saves time and resources, by keeping the image data in GPU memory and saving in CPU-GPU and GPU-CPU communication overheads. After the features are extracted, they are shared with the other n machines (step 2), which then execute feature matching and compute the graph (step 3), outputting a graph (or several, depending on the amount of computer vision algorithms used) similar to what can be observed in Figure 1.2.

The usage of several computer vision algorithms, and the consequent output of several graphs (which contain completely different images features and scores), enables users to

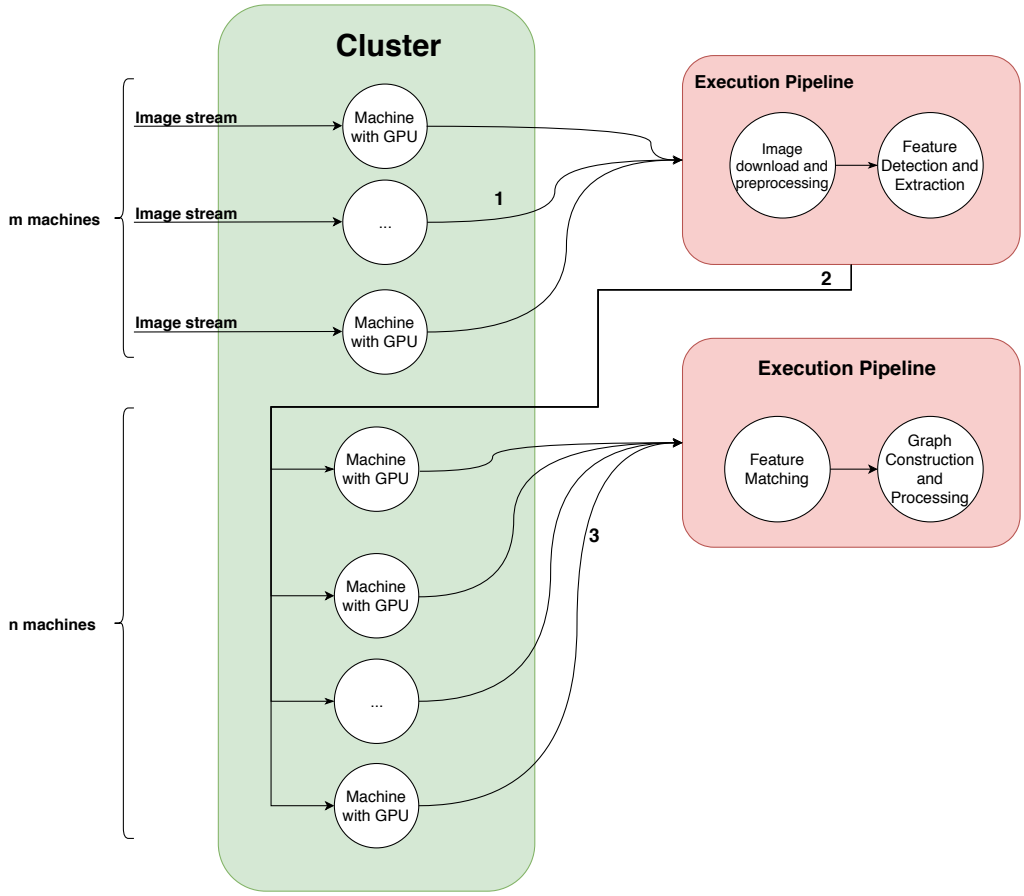

Figure 1.1: *Distributed Pipeline*

take advantage of the diversity of computer vision algorithms for different use cases. For instance, some algorithms may be better at detecting objects in images, while others may be more suited towards detecting humans. Processing and making this data available all in one, parallel, execution instead of multiple executions (one for each algorithm) saves the user time, since the image is already downloaded, preprocessed and loaded on the GPU.

The querying process is identical to the one just described. A machine downloads the query image, pre-processes it and performs feature extraction. It then shares the image features with the machine (or machines) that contain the previously computed graphs. These machines, in turn, perform feature matching between the query image features and the previously computed features. They use this result in order to index the query image in the graph. Once this indexing process is complete, a graph primitive (such as Dijkstra's algorithm) is executed, in order to compute the relevant query (*e.g.*, find the n closest matches).

It is also possible to perform a query using an image that was already processed and is present in the database. In this case, the first step (query image download and feature extraction) can be omitted, as it was already executed previously by the main pipeline of our library. All that is necessary, then, is to execute the graph primitive relevant to the

query on the previously computed graph (or graphs). For instance, if we were to perform a query on the (tiny) graph present in Figure 1.2, in order to obtain the most similar image to image 1, we would learn, through Dijkstra's algorithm, that such an image is image 3 (considering that the lower the score, the more similar two images are). The usefulness of Dijkstra's algorithm becomes more obvious with larger graphs, as they may contains hundreds of thousands of nodes and hundreds of millions of edges.

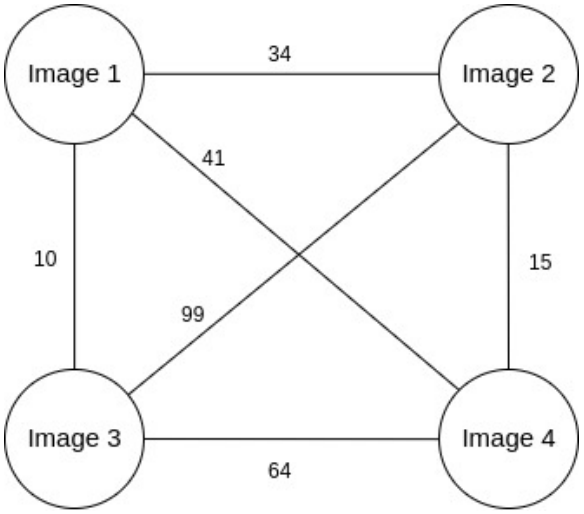

Figure 1.2: *Example graph*

To conclude, there are several different configurations that our solution can be executed in (*e.g.* varying m and n), each with different advantages. We may require more diversity in our outputted graphs, or we may require sheer speed for a single computer vision algorithm. This means our library needs to be easily configurable and parametrizable, in order to meet the user's needs. Besides the different pipeline configurations, there are also several different feature detection algorithms, graph construction and processing libraries that can be used. The intent is to develop the library so that these components are easily interchangeable and, where they are not, we need to make sure that the ones used are the most efficient in order to achieve a better, faster solution.

1.4 Contributions

This thesis presents the following contributions:

- A solution that is able to process images and extract some number of features from them (using different algorithms for different features and use cases).
- A system that can compare images against each other and compute graphs for each set of features extracted, with edges that contain the similarity between the two connected nodes (images).

- A system that is able to take advantage of the graphs in order to process query images to determine images similar to them, both visually and in time.
- A system that outputs graphs that contain diverse information about all images processed, allows comparisons between them and enables users to take advantage of them for numerous use cases.
- A scalable system, capable of taking advantage of clusters vertically (by harnessing the power of each cluster node as much as possible) and horizontally (by harnessing the power of every single node), as well as take advantage of the power of GPU-accelerated algorithms.
- Several experimental evaluations detailing the performance of our solution with large image datasets.

1.5 Document Structure

In this chapter we introduced the problem this thesis attempts to tackle and the contributions it intends to make. We presented the motivation behind our work, the problems that it must face and their solutions.

In Chapter 2, we introduce the architecture of the NVIDIA GPU and how it benefits our work, along with an introduction to GPU programming using CUDA. We also present the current state of the art for many different modules, libraries and algorithms our solution intends to take advantage of such as feature detection, description and matching, graph construction and processing, and distributed computing. Chapter 2 finishes off with a comparison of modern libraries similar to the one we intend to build.

In Chapter 3 we cover a top-level view of our system, its architecture and how it works.

Chapter 4 covers through the implemented solution thoroughly, entering into detail about every section and nuance of the system. Leaving these chapters, the reader should be able to understand the logic behind our choices and have an understanding of how to build such a system.

Chapter 5 details the experimental evaluations of our library, measuring its performance and precision.

Finally, Chapter 6 finishes off by extracting conclusions derived from our work, as well as intentions and possibilities regarding future work on the system.

Chapter 2

State of the Art

This Chapter presents the current state of the art in the areas this thesis covers. It starts by presenting the architecture of GPUs and their programming. It then covers several computer vision algorithms, how they work and their advantages and disadvantages. We also provide a preliminary benchmark for these algorithms, in order to compare them. We then cover feature matching techniques as well as GPU accelerated solutions for them. Afterwards, we detail and compare GPU graph processing libraries and distributed stream processing libraries. We finalize by presenting several image processing systems similar to our own.

2.1 GPU Architecture and Programming

Graphics Processing Unit (or GPU, for short) is a specialized, programmable electronic circuit designed for the rendering of images, animations, videos and everything related to computer graphics. It is optimized for the execution of vector transformations, matrix calculation and operations, and floating point operations. GPUs are frequently used in gaming systems, animation rendering and video processing, being highly efficient at those tasks. They are good for and designed to process everything related with computer graphics. GPUs are in everyday systems, from personal computers to mobile phones and gaming consoles. These circuits have a highly parallel architecture, making them more efficient than many modern CPUs at the computation of certain, highly parallelizable, algorithms.

The application of GPUs for general purpose computing is named General Purpose Computing on Graphics Processing Units (GPGPU) and is a fast growing field, applicable in chemistry, fluid dynamics, computer vision, cryptography, blockchain technology, the

financial sector and many more. Several industries now take advantage of the computational power of GPUs, due to the fact that a GPU typically has thousands of cores while a CPU only has a few, as Figure 2.1 illustrates. GPUs have been designed, since their conception, for highly-parallel intensive tasks, which makes them perfect for certain types of computations, and many times faster than CPUs. In response to the emergence of the necessity of using GPUs for general purpose computation, some libraries, APIs and frameworks emerged that enable the programming of GPUs in a much simpler fashion, such as NVIDIA's CUDA [25] (for NVIDIA graphics cards) and OpenCL [36] (which supports a broader range of GPU manufacturers including NVIDIA and AMD), which abstract the programmer from low-level GPU details.

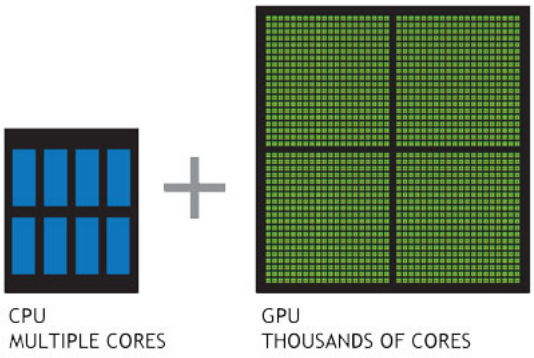

Figure 2.1: *CPU vs GPU cores*[1]

2.1.1 NVIDIA Architecture

There are several NVIDIA graphics cards microarchitectures that were developed over time, however, we cover only the basics that are transversal to all modern microarchitectures and explain how they relate to the execution of parallel programs and algorithms, using as example the most modern NVIDIA microarchitecture named Pascal [34].

The architecture of a Pascal GPU can be observed in Figure 2.2. It breaks down into several simpler components, of which we name a few:

- Graphics Processing Clusters, composed of 10 texture processing clusters (TPCs)
- Texture Processing Clusters, composed of 2 streaming multiprocessors (SMs)
- Streaming Multiprocessors
- Memory controllers
- PCIe host interface, to communicate with the host device

[1] Image taken from the NVIDIA website at http://www.nvidia.com/object/what-is-gpu-computing.html

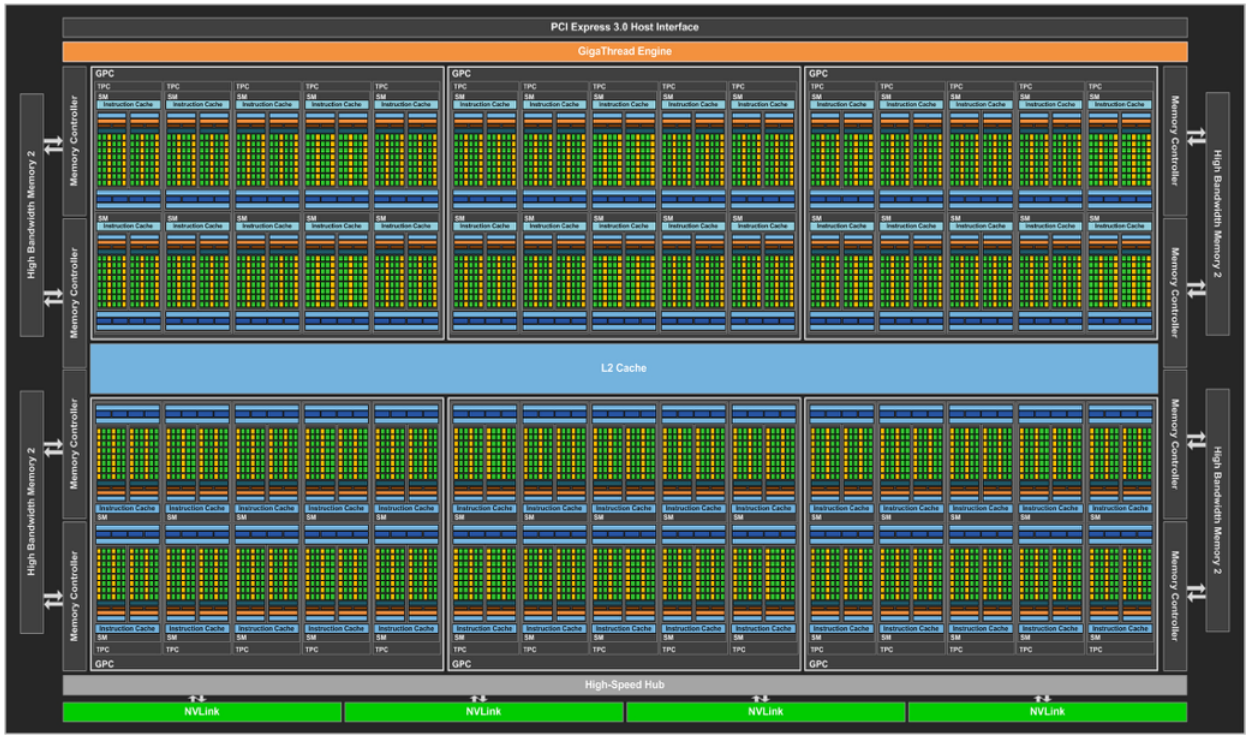

Figure 2.2: *Pascal microarchitecture GPU, taken from [34].*

NVIDIA GPUs process instruction streams in groups of 32 threads called warps. Each warp shares an instruction counter and every thread in the warp executes the same instruction simultaneously. If threads within a warp take different execution paths (*e.g.* in conditional statements), warp divergence occurs and performance is significantly reduced. A group of 1 to 32 warps is called a thread block, and threads within a block have access to a piece of shared memory. Work is partitioned by assigning it to blocks and threads within blocks through thread and block IDs. Each block (occasionally, and in some architectures, more than one) is executed on one of the streaming multiprocessors (SM) of the GPU.

The architecture of the streaming multiprocessor (SM) can be observed in figure 2.3. An SM consists of several cores (which perform arithmetic, logic and floating point operations), special function units (SFUs, which execute special functions such as sin, cosine and square root), load/store units (LD/ST, which calculate source and destination addresses), instruction buffers which contain instructions for the processing units, and warp schedulers and dispatch units that issue instructions to warps. Finally, the SM has a 64KB shared memory and L1 cache, allowing threads within thread blocks to cooperate between them and cache data. The particular SM shown in figure 2.3 contains 64 cores and is part of the Pascal microarchitecture.

As can be observed, the architecture of the GPU is designed to run thousands of instructions simultaneously. It contains thousands of cores for that purpose that, even though they run at a much lower clock speed than most modern processors (1328 MHz

Figure 2.3: *Pascal streaming multiprocessor, taken from [34].*

in the Pascal microarchitecture) and have lower throughput and higher latency than any modern CPU core, they can execute many simple computations very quickly and achieve massive parallelism that can not be matched by any CPU on the market.

2.1.2 Programming: CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA for general purpose computing in graphics processing units. It enables programmers to significantly speed up computing applications by offering them a platform that allows them to run programs developed in C, C++, Fortran, and other languages, on the GPU. It removes the need to express computation through the programming of shaders as was needed before frameworks such as CUDA and OpenCL first emerged and has the great advantage of not having any overheads, as NVIDIA's GPUs are developed to run CUDA code, and they excel at it. This makes CUDA-capable NVIDIA GPUs highly efficient at both the computation of graphics and the general purpose computing tasks that we wish to execute.

CUDA programs work by calling parallel kernels (each kernel is a unit of work) which are split into grids (each grid is a group of several thread blocks), and then executed in parallel across several thread blocks (containing warps, as discussed previously in

Section 2.1.2) within a grid.

Typically only one kernel executes at a time, except in very modern GPUs, where spare resources are assigned to a second kernel whilst the first is running. Kernels can also execute in parallel, where blocks from several kernels are executed simultaneously in a grid. It is also possible to use certain ordering primitives in order to implement absolute ordering between kernels.

In sum, groups of threads (warps) are organized into thread blocks (which are executed on SMs) and blocks are organized into grids. The hierarchy of threads employed by CUDA can be observed in Figure 2.4. As can be seen, each single thread has a private local memory used for function calls and other functionalities usually related to the programming language being used. Threads are organized into blocks which have a per-block shared memory. This shared memory allows threads within blocks to cooperate, however, threads that belong to different blocks can not cooperate. This means we can only achieve thread cooperation at the block level, and rarely at the whole program level. This is why it is best to run programs with low dependencies between calculations, as cooperation can be expensive in GPUs. Global memory (in the case of the NVIDIA Pascal Titan X, 12GB in size and with higher bandwidth than typical CPU memory) is used by the kernels running on grids (and blocks within grids) for communication and sharing computation results.

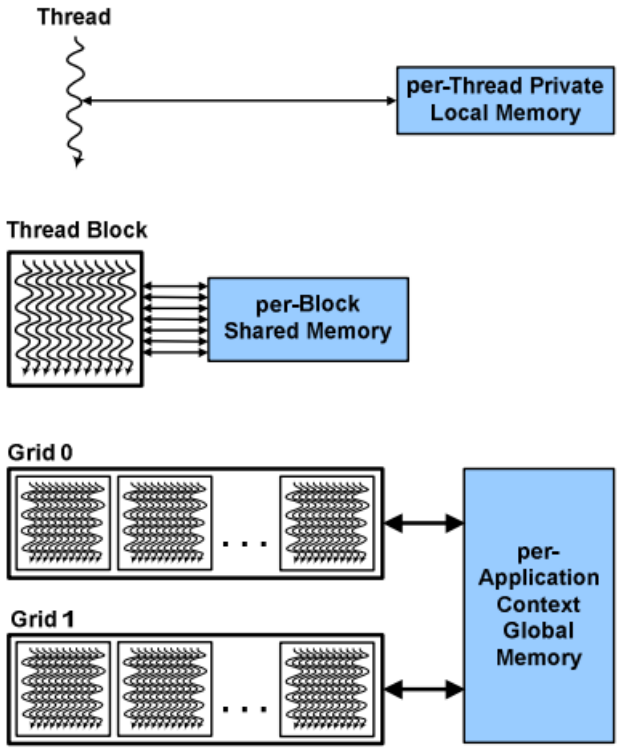

Figure 2.4: *CUDA Thread Hierarchy* [2]

[2] Image taken from the Fermi Whitepaper [19]

The execution model is detailed in figure 2.5. As we can see in this example, the host specifies two kernels to execute. The first kernel is assigned to Grid 1, which contains 6 blocks of threads while the second kernel is assigned to Grid 2. The vertical black arrow on the left represents time, meaning kernel 1 executes before kernel 2. Each block contains, in this example, 15 threads. Note that each block (or more, depending on shared memory usage per block) is executed on a streaming multiprocessor and each thread within a block is executed on a CUDA core within the SM (illustrated in figure 2.3). It is important to consider that these details are abstracted when writing CUDA code, and the programmer does not need to worry about such low level details unless he wants to fine-tune his code to be extremely efficient. Most of the time, a simple CUDA implementation is sufficient for the objective of accelerating computation and achieving much faster performance than would be achievable on a CPU.

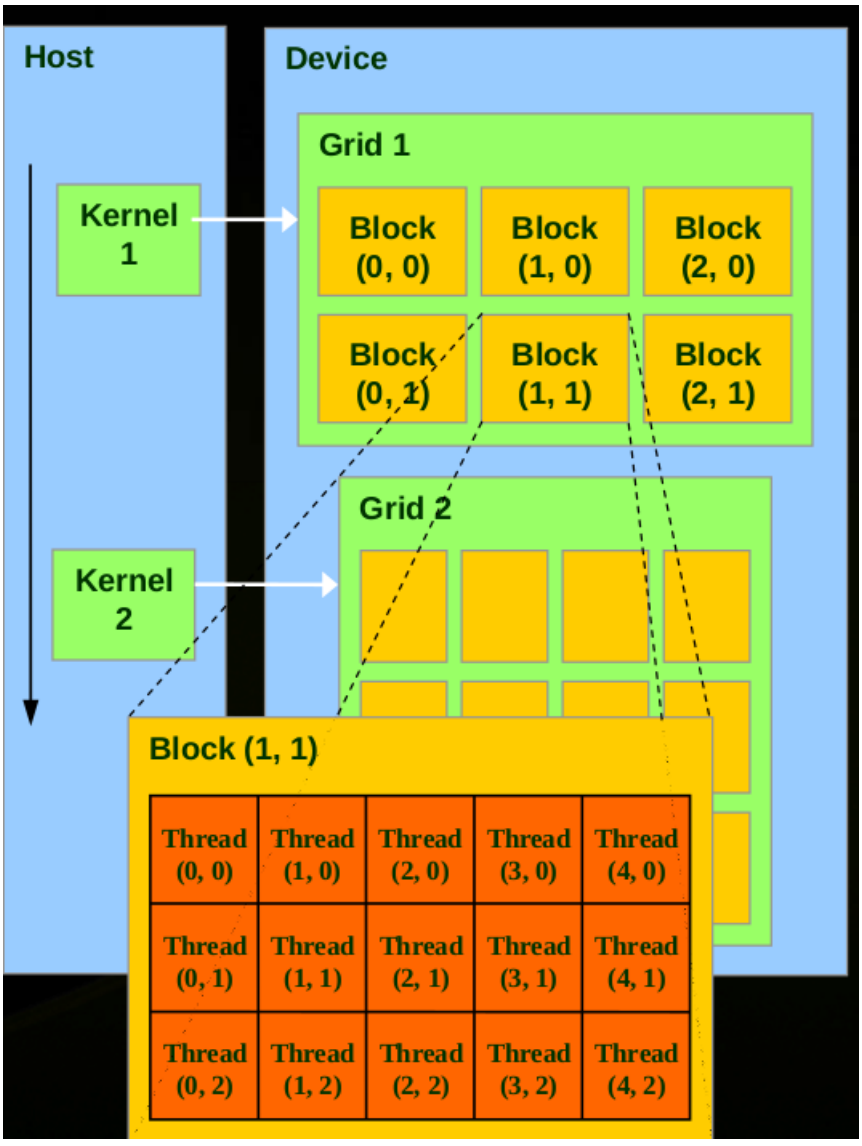

Figure 2.5: *CUDA Execution Model*

2.2 Image Feature Detection and Description on GPUs

There are several algorithms to detect visual features and describe images, with varying performance and precision. Some algorithms describe images in a more precise way, while others execute faster. Some are more adequate to detect human faces, while others excel at object detection. If we require very precise features and image descriptors, in order to avoid similar but different images to be detected as the same image, there are algorithms that provide that. However, if we want our solution to have good performance, and do not mind loss of precision in the image feature vector, we must choose an algorithm that meets those requirements.

Since our objective is to build a graph capturing several image similarities and we intend to process very large amounts of images, precision can be relaxed to allow for faster extraction. In order to achieve even better performance, we consider the GPU-Accelerated versions of some of these algorithms.

The algorithms shown in this section are some of the most important algorithms that detect and extract features from images. An image feature vector consists of a vectorial representation of the image, allowing us to considerably reduce the image's size and still be able to recognize and compare it with different images, which is useful when considering the amount of images we need to process and store in memory. We only cover algorithms that are relevant to our work.

In the coming subsections, we briefly describe each algorithm and how it functions. We will process images using a selection of these algorithms in order to extract visual features. Later, using these, we will be able to compare images and calculate how similar they are.

2.2.1 OpenCV

Open Source Computer Vision (OpenCV) [3] is, an open-source computer vision and machine learning software library. It contains several algorithms for computer vision, from modelling 3D objects to detecting features from images. It has a very big community, number of contributors and number of users, making it a library with great support and number of features. Additionally, it contains high support for CUDA, enabling us to accelerate these algorithms' executions on our GPU cluster, without the need to develop specialized CUDA kernels. Although the CUDA interface is still being developed as of the moment of writing this document, support for most of the algorithms we tested is widely available.

Considering the focus of this thesis, we take advantage of the algorithms and libraries provided by OpenCV in order to detect and compare image features. We explore several of these algorithms in the subsequent sections.

2.2.2 Histogram of Oriented Gradients

Histogram of Oriented Gradients (HOG) [8] uses, as the name suggests, a histogram of gradients to describe an image. It first computes the vertical and horizontal gradients of an image. Afterwards, the algorithm calculates the magnitude and direction for each gradient, where the direction points to the direction of the change in intensity and the magnitude points to how big the actual change is. Calculating gradients allows us to know the locations of the images where a sharp change in intensity occurs, therefore allowing us to identify the edges of the image, which are typically very useful for identifying visual signatures. An example of the gradients calculated from changes in intensity can be observed in Figure 2.6.

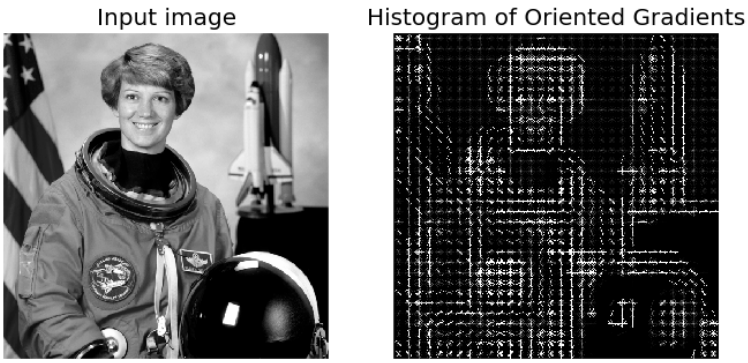

Figure 2.6: *HOG Visual Features*[3]

After the gradients are calculated, the image is then divided in 8x8 cells, and a histogram of gradients is calculated for each of the cells. The histograms have 9 bin values, each bin corresponding to an angle, as explained below.

The histogram is calculated and represented in an array where each index of the array corresponds to a bin of the histogram which, in turn, corresponds to the angle, in degrees, of the gradients in that cell. The value of each bin corresponds to the contribution of each gradients' magnitude. An example histogram can be observed in Figure 2.7. The more value the histogram has in a certain bin, the higher the magnitude of the gradient is for that angle. The bins correspond to the angles 0, 20, 40, 60, 80, 100, 120, 140 and 160. It is important to note that if the angle is between 160 and 180, it contributes proportionally to the 0 and 160 degree bins [22].

The last step consists of calculating the final feature vector for the entire image. All the vectors (histograms) previously calculated are concatenated into one vector of 3780 dimensions. Afterwards, we are left with a histogram describing the image itself. It describes the images' changes in intensity by distributing them by the angle of the direction

[3]Taken from the scikit-image website at http://scikit-image.org/docs/0.11.x/auto_examples/plot_hog.html

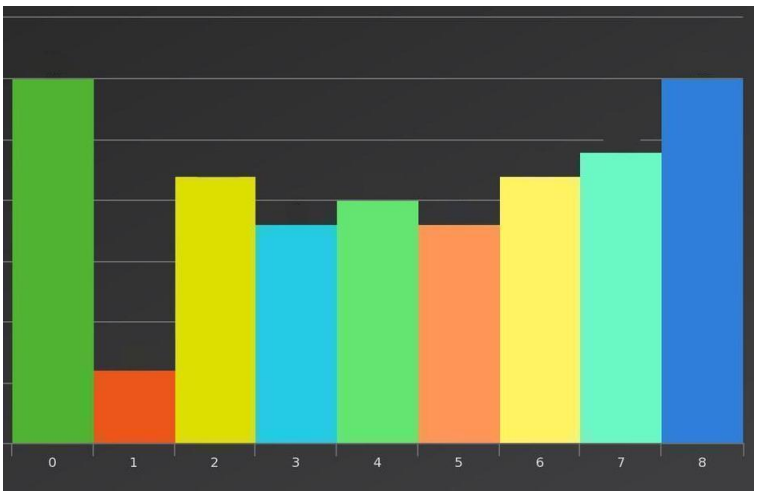

Figure 2.7: *HOG Histogram*

of the intensity.

This algorithm is typically used for human detection, *i.e.*, face detection, pedestrian detection. It can also detect humans in different poses, from different angles. Due to its precise ability to identify sharp edges in images, it is one of the better performing algorithms for that purpose.

2.2.3 Scale-Invariant Feature Transform

Key points represent locations of interest in an image. These are points that stand out and, when combined with each other, are able to describe the image visually. Key points are important for image description because they are usually unique, and enable differentiation between images. Image key points are associated with visual corners, *i.e.*, the point where the directions of two edges change, meaning the gradient has a high variation. Several of these key points enable us to uniquely identify an image. Figure 2.8 shows us the key points for an example image. The size of the circle represents the size of the key point and the line within each circle represents its orientation.

A big problem that Scale-Invariant Feature Transform (SIFT) addresses is the fact that some algorithms are not scale-invariant. This means that a corner may not be a corner if the image is scaled. If we zoom into a corner enough, it may become flat and, thus, not be a corner any more (or atleast not be detected as one).

In order to address this, the authors in [20], developed SIFT. The algorithm works by extracting key points and computing their descriptors. SIFT first applies the scaling mechanism, which consists of scale-space filtering. The images are searched for local extrema, *i.e.*, one pixel in the image is compared with several neighbours in higher and lower scales and, if it is a local extrema, it is a potential key point and is best represented

[4]Image taken from the OpenCV Documentation at https://docs.opencv.org/3.3.0/da/df5/tutorial_py_sift_intro.html

Figure 2.8: *SIFT Key points* [4]

in a certain scale. So, in sum, a pixel is tested for potential candidacy of being a key point in several different scales, thus achieving scale-invariance, as the name promises.

Once the potential key points are found, they are first filtered for noise and then assigned an orientation, in order to make the algorithm rotation-invariant (*i.e.* resistant to rotations of the image).

The key point descriptor is created for each key point, outputting a histogram of 128 bin values. Finally, this set of key point descriptors is what enables us to identify and compare the image.

The features extracted are both distinctive and precise, making this a robust algorithm, able to compare images correctly with high probability. The only issue with this algorithm is the fact that it is slower than most algorithms, which is what the next algorithm addresses.

2.2.4 Speeded-Up Robust Features

In order to address the performance issues in SIFT, the authors in [1] created a speeded-up version of the algorithm. They developed a new method for the scale-space filtering which, in short, has a different approximation for the Laplacian of Gaussian (used for scale-space filtering in edge detection) than the one used in SIFT. It is not only a faster computation in itself, but it can also be computed in parallel. Additionally, it has several optimization parameters, such as bypassing the part of the algorithm that makes it rotation-invariant, since many applications do not require it.

The SURF feature descriptor has 64 dimensions, but can be extended to 128. This is

also an optimization, as fewer dimensions make the algorithm faster but the features less distinctive. However, it is a trade-off some applications may want to make. Several other optimizations are detailed in the paper, but the basis of SURF is a lot of improvements and parametrizations in each of the SIFT steps, resulting in an all-around faster algorithm, albeit less precise.

2.2.5 Features from Accelerated Segment Test

The motivation behind the Features from Accelerated Segment Test (FAST) algorithm [28] derive from the fact that feature detectors as the ones showed previously are not particularly fast enough for a real-time application.

This algorithm works by select a pixel p, to be tested as a key point candidate, and calculating its intensity I_p. Considering a threshold value t and a circle of 16 pixels around pixel p, the algorithm considers pixel p to be a corner if there exists a set of n (in the paper, n=12) contiguous pixels that are brighter than $I_p + t$ or darker than $I_p - t$.

In order to speed up this test, the algorithm only examines four pixels (at opposite sites of the circle). If atleast three of these four points are brighter than $I_p + t$ or darker than $I_p - t$, then the pixel is considered a candidate to be a corner. After determining potential candidates, the algorithm runs these computations again for all points of the circle for each candidate, in order to determine the final corner pixels.

Some issues (such as redundant features) arise from this methodology, however they are solved using a machine learning approach and non-maximum suppression, described in the paper.

It is important to note that this algorithm only detects key points, it does not compute descriptors. In order to compute a feature descriptor using this algorithm, we must use another algorithm in conjunction with FAST for the computation of the descriptor. The key points detected by FAST can be observed in Figure 2.9.

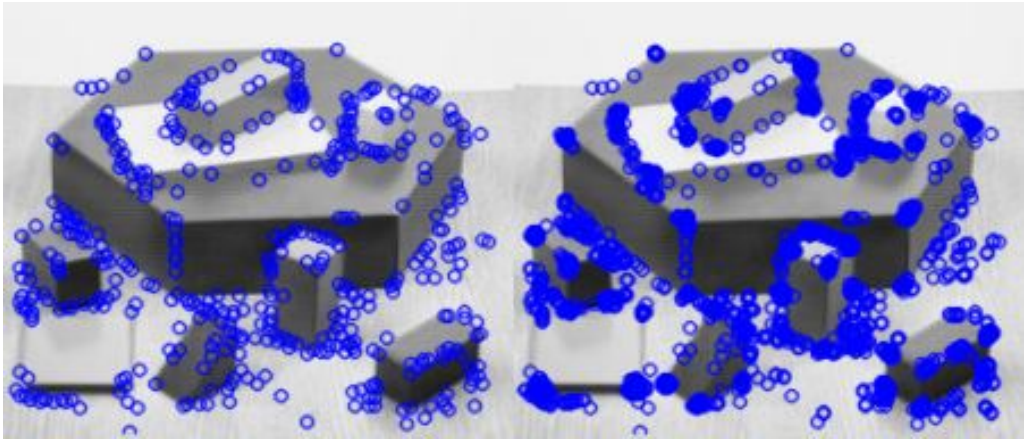

Figure 2.9: *FAST Keypoints*

[4]Image taken from the OpenCV Documentation at https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_fast/py_fast.html

In sum, this algorithm is significantly faster than the others described in regards to the key point computation, but it is not as precise and may suffer from images with a lot of noise. Also, it is still dependant on another algorithm in order to compute a feature descriptor, as it is only a key point detector.

2.2.6 Vector of Locally Aggregated Descriptors

The vector of Locally Aggregated Descriptors (VLAD) [18] algorithm consists of, as the name suggests, aggregating local descriptors in a vector. The main focus of the development of this algorithm was on three constraints: search accuracy, efficiency and memory usage.

The algorithm first computes a set of k visual words $C = \{C_1, ..., C_k\}$ (*i.e.*, neighbourhoods or small parts of the image where features are extracted from) using k-means. Then, each local descriptor x is associated with its nearest visual word and the difference between the local descriptor and the associated visual word is computed. This difference characterizes the distribution of the vector with respect to the center. The vector is then computed as the sum of all the differences between the local descriptor and the associated visual word for each of the k neighbours.

This vector then serves as input in order to compute a code of B bits, encoding the image. Several optimizations are performed in order to produce this encoding (such as dimensionality reduction and indexation), better described in the original paper.

This algorithm is very accurate and fast in searching operations, being able to search a 10 million image dataset in only $50ms$, achieving its design goals of memory usage, search accuracy and efficiency.

2.2.7 Deep Learning Visual Features

The algorithm developed by the Visual Geometry Group (VGG) [31] relies on a convolutional neural network, trained to classify images. Using their algorithm and a pre-trained network containing millions of images, we can extract visual features from images. The algorithm outputs a 4096 dimensional feature vector, containing the neural network activations.

As this algorithm relies on an external neural network, it is only as good as the network itself, and is dependant on the images that it was trained with. Also, the activations are normalized, enabling us to use the feature vectors as as generic features for image similarity calculation.

2.2.8 Binary Robust Independent Elementary Features

Binary Robust Independent Elementary Features (BRIEF, for short) [4] is an algorithm developed with the intent to solve memory constraints introduced by other computed vision algorithms. SIFT, for instance, uses a 128-dimension feature vector (consisting

of single-precision floating point numbers) which means each SIFT descriptor occupies around 512 bytes in memory. This does not seem like much, but we if consider processing 10 million images, for example, it amounts to more than 5GB. This is a very large amount to hold in memory and becomes a big constraint in programs that wish to process and match large amounts of images.

BRIEF takes advantage of the fact that not all dimensions of a feature descriptor are needed to perform feature matching. It is possible to compress these descriptors, using methods like locality sensitive hashing. These compression methods convert descriptors from floating point numbers into binary strings. In order to compare such binary strings, for feature matching, it is possible to use a norm such as Hamming distance. BRIEF uses a unique way (which is detailed in the paper) to find binary strings directly, without the need to find descriptors first. This means the memory issue is addressed directly, as we do not require computing a feature descriptor first and only then compress it.

It is important to note that BRIEF is only a feature descriptor. It does not detect key points. In order to use BRIEF, we have to pair it with another algorithm that can perform feature detection (such as SIFT, SURF or FAST, for instance).

This algorithm is a fast and light feature descriptor, while also providing good precision when matching. It is suitable for libraries that intend to process large amounts of images and are constrained by memory.

2.2.9 Oriented FAST and Rotated BRIEF

Oriented FAST and Rotated BRIEF (ORB) [29] is an algorithm that, essentially, consists of a fusion between the FAST key point detector (previously detailed in Subsection 2.2.5) and the BRIEF descriptor (Subsection 2.2.8). This is, however, not a simple concatenation of these algorithms, as it employs several strategies to improve precision. First of which is tweaking the key point detection so that it is rotation invariant (as FAST does not provide this natively). Additionally, it improves the precision of BRIEF descriptors by tweaking them to better define the orientation of the key points.

This algorithm takes the best of the FAST detector and BRIEF descriptor and attempts to solve or lessen the impact of their downsides. ORB is much less memory intensive than algorithms such as SURF and SIFT and provides a much faster matching solution, although precision is not nearly as good as in these algorithms.

2.2.10 Preliminary Benchmark

In order to properly compare the performance of the above algorithms, we created a simple testbed that enabled us to benchmark and test the algorithms. We only considered algorithms supported by OpenCV, as they are easier to implement, well established, and support for them is widely available. It also enables us to easily switch between different algorithms. Additionally, OpenCV enables image comparison (*i.e.* feature matching) and does it in a handful of different and efficient ways. The tests were run with a library

Table 2.1: OpenCV Feature Detection and Description Algorithm Comparison

Algorithm	**Feature Extraction GPU (ms/image)**	**Feature Extraction CPU (ms/image)**
SURF	16.3	41.3
HOG	5.2	19.9
ORB	34.8	48.8

of 500 images from the YFCC100M dataset [35], measuring the average time it took to detect and describe the features from each of those images, for each algorithm. We do not consider the download time for the images or the GPU upload time in this test, in order to focus more on the execution of the algorithm itself and due to the fact that these overheads are inevitable and do not depend on any algorithm. The results displayed are an average of three executions. Results for the algorithms executed on the CPU are also displayed. These are aggregated in table 2.1. The algorithm represents both the feature detector and descriptor, and the times displayed consist of the time it takes to detect key points in an image and generate a feature vector to describe the image. As previously explained, this vector is the vectorial description of an image and enables us to compare images with each other.

The testbed consists of an Asus N751JX laptop, with an Intel Core i7 4720HQ Processor with a clock speed of 2.6GHz (turbo boost up to 3.6GHz) and a NVIDIA GeForce GTX 950M with 2GB of VRAM.

As we can observe from the table, it is clear that running these algorithms on a GPU incurs in a significant speedup over execution on the CPU. In all three algorithms that we tested, the benefit of using the GPU is clear. Obviously, any small microsecond gain is very significant in libraries that intend to process millions of images. Out of these three, HOG executes faster, although its feature matching process is typically slower, and its features are much more memory intensive than the other algorithms tested. The ORB algorithm has a much higher extraction time than its peers, which is expected, as its memory saving techniques incur in a higher processing time. In this case, we are trading off processing time for less memory usage, while in other cases (such as the HOG algorithm) we benefit from speed while sacrificing memory usage. SURF is a more balanced algorithm. Although it is less precise than HOG, its features are less memory intensive. It is important to note that each of these algorithms excel at different tasks and have different use cases. For instance, and as described in the previous subsections, HOG is more adequate for human and face detection, while SURF and ORB are algorithms more suited towards object detection.

Note that this, as the subsection title suggests, is a mere preliminary test. It only intends to evidence the fact that executing these algorithms on the GPU has a great advantage over execution on the CPU. It also provides a quick, rough comparison of the tested algorithms, to support the explanations provided in the previous subsections. More detailed tests and parameters will be provided in Chapter 5.

In conclusion, in order to meet speed requirements, for instance, one may use the HOG algorithm. However, if memory is an issue, it may be wise to use an algorithm such as ORB, or even SURF to meet a balance between both. Nonetheless, usage of OpenCV entails in a great advantage: the ease of interchangeability of algorithms on the fly enables great adaptability to different constraints.

2.3 Feature Matching

Feature matching is an essential part of our work, as it is necessary to be able to compare images between each other and calculate how similar they are. It is also a necessity for us to query the system with an image and be able to tell if that or a similar image exists. Being able to tell if two images are the same or, at least, similar is very useful, as we went over in Chapter 1. Also, the ability to detect if a certain object is in a certain picture (*e.g.* a human face among a picture of a crowd of people) is a very useful feature with many use cases. This is the functionality provided by feature matching. After computing the feature descriptors of images, we need to run a matching algorithm between them, outputting matches between images.

2.3.1 k-Nearest Neighbours

k-Nearest Neighbours is a machine learning algorithm used for classification and regression. The algorithm consists of a training phase, where it is fed examples of multidimensional features, each with a class label. Afterwards, in the classification phase, the algorithm classifies an unlabelled feature vector by assigning it the label that is most frequent among its k-nearest neighbours. There are several metrics to compute the k-nearest neighbours, the most common being Euclidean distance. It is a very simple and powerful algorithm.

The k-Nearest Neighbours algorithm can be applied to computer vision, as the result of feature extraction is a multidimensional feature vector that describes the images. If, for example, we want to classify images and be able to tell pictures that have cats in them from pictures that have dogs in them, k-NN can achieve that. Additionally, if we want to compare any two images between them and be able to tell how similar they are, we can also request the help of the k-NN algorithm. In this case, in conjunction with what is called brute-force matching, k-NN takes the descriptor of one feature of the first image and matches it to all the other features in the second set and, using a distance metric as explained before, returns the k closest matches. Figure 2.10, obtained from [11], illustrates this well. Consider the red cross to be the feature of the image we want to compare for similarity and the blue dots to be features of one of the training images (*i.e.*, the images already in our database). In the case of the image, $k = 3$, which means the three blue dots inside the circle are the 3 features that best match the feature of the query image being tested.

The author of SIFT proposed in [21] a ratio test to evaluate the similarity of two images. It is necessary in order to discard features that are not useful and possible false matches, such as ones generated from background clutter in the image. It takes the ratio between the distance of the closest match and the distance of the second-closest match (usually paired with k-NN, with $k = 2$). This discards false matches well because there is usually a number of other false matches within similar distances, due to the high dimensionality of features. In the paper, the author shows that by rejecting matches with a ratio greater than 0.8 we can eliminate 90% of the false matches while only discarding less than 5% of correct matches. Applying this test to the matches obtained by k-NN across all features of the query image, we are left with the matches between two images and we can tell with a good degree of confidence whether or not the images are similar. We can also compute how similar they are by designing a score that measures the ratio between good matches (the ones filtered by the ratio test) and the total number of matches (before the filtering), and even if they contain the same objects.

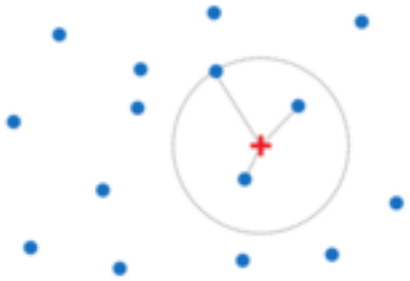

Figure 2.10: *k-NN Illustration*

k-NN Illustration with k = 3. The red cross is the point being tested, while the blue dots are the points from the training data.

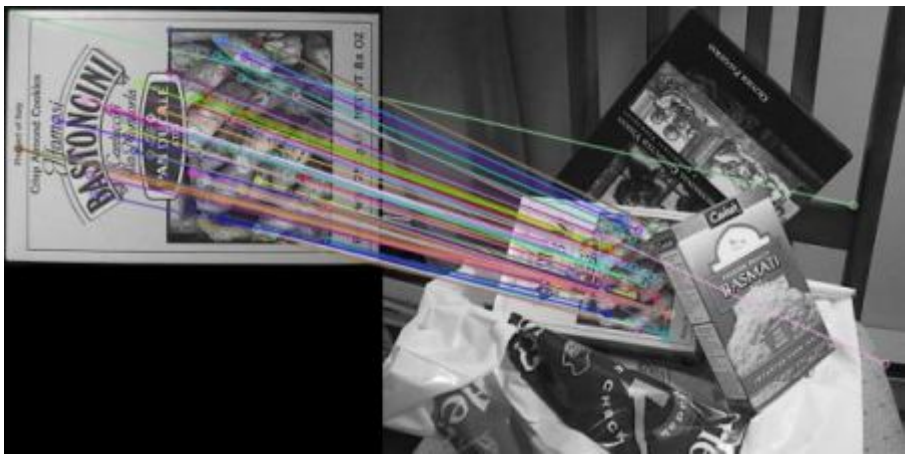

Figure 2.11: *Brute-force matching with k-NN*

Each line is drawn between the two features in both images that more closely match.

Note that features of images are usually corners in most algorithms, as explained in section 2.2. We can see this matching in action in figure 2.11, taken from the OpenCV [3] website. The algorithm picks a feature from the query image (left) and executes k-NN as described above, comparing it with the features from the training image (right). It selects the two closest features ($k = 2$) and performs the ratio test, in order to filter out most incorrect matches. It does this for all features of the query image, finding all matches. Each line is a match, *i.e.*, it is drawn between the two features of both images that more closely match. As we can see, brute-force matching with k-NN finds a lot of matches between the two images (albeit still containing some noise) and can correctly assume their similarity. In this case, the algorithm is finds an object (the box on the query image on the left) in an image with a group of objects (the training image on the right).

2.3.2 GPU-Accelerated k-Nearest Neighbours

Following the line of thought we have been following so far, we want to accelerate our solution wherever we can and feature matching is a computationally intensive step. Obviously, this also means using a GPU-Accelerated implementation of k-Nearest Neighbours, as it is a highly parallelizable algorithm, which means it can be significantly sped up by doing so.

OpenCV features a CUDA implementation of the brute-force k-nearest neighbours matching algorithm, which significantly speeds up the computation. It is easy to use, like most CUDA libraries in OpenCV, and it can be easily interchanged with the CPU version, if there is need to do so. There is also a lot of recent work on the topic of executing k-NN on GPUs.

The approach in [11], implemented using CUDA, works by specifying two CUDA kernels where the first calculates the distance matrix between the query points and the training points, and the second sorts the distance matrix, retrieving the k points closest to the query point. The big optimization is in reformulating the way the distance matrix is calculated by optimizing the distance calculation using linear algebra and computing it using the CUBLAS library (CUDA implementation of the highly optimized BLAS linear algebra library). The experiments run by the authors of that work show that the CUBLAS implementation is up to 62X faster on SIFT feature matching than the Approximate Nearest Neighbour (ANN) C++ library.

Gieseke et al. [12] combine k-dimensional trees and GPUs by proposing a variant of k-d trees called buffer k-d tree, showing good speedup against other k-d tree GPU implementations and brute-force k-NN.

The work in [26] attempts to reduce the search space by partitioning datasets into several groups in order to cluster similar items and using locality-sensitive hashing (LSH) to speed up the query process. The GPU implementation shows a 40X speedup over other LSH single-core CPU approaches.

Another approach, called Sweet k-NN [5], tries to combine the strengths of optimizing

redundant distance computation (*i.e.*, reducing the search space) and executing massively-parallel computation on the GPU. It strives to achieve balance between minimizing redundancy and preserving regularity (which means better performance on GPUs and linear algebra libraries). Sweet k-NN is a very recent approach and shows up to 44X speedup over the work in [11] in some datasets. As far as we are aware, this is the current state of the art in the area of GPU-accelerated k-nearest neighbours algorithms.

The key points that are important to our work is the ability to store the feature descriptors correctly in GPU memory in order to efficiently perform comparisons and reducing the search space, in order to perform less redundant computation and result in overall lower execution time, while maintaining precision. The OpenCV implementation is easy to use and to integrate with our work, since features will already be extracted using OpenCV. It is the simplest solution, while also having GPU support. We also consider using Sweet k-NN in the future, supporting the modularity design goal of our solution, and being able to easily change between implementations.

2.4 GPU Graph Processing

In the context of our work, after image features are computed, we need to organize them in a graph. Each node in the graph corresponds to an image (its features) and the edges between nodes correspond to the similarity between them. A node has n edges for each of its n neighbours, *i.e.*, its n most similar images. The construction of this graph is a challenge due to its size, the actual computation, the memory constraints (how to store the graph in multiple GPUs and multiple machines) and, most importantly, the necessity of executing queries efficiently and quickly on the computed graph.

After the computation, the output is a graph as big as the number of images processed and containing as many edges as there are similar images, meaning a very large dataset will output a very large graph. This graph is key to our design goal of being able to query images for their similarity, and the way to achieve that is through running primitives on the graph that allow us to extract information from it.

Once again, we turn to GPUs in order to solve all of these challenges, from graph construction to the execution of primitives (such as breadth-first search) on the constructed graph. Like in the case of image feature detection and description detailed in Section 2.2, graph processing benefits greatly from the parallelism offered by GPUs. There are several libraries that support high-performance graph processing on the GPU and we will examine the state of the art solutions in this section and explain which fits our work better.

2.4.1 Solutions

There are several ways to process graphs and, using the criterion used by Wang et al. in [38], we list the different types of systems, their advantages and disadvantages

subsequently:

Single-node CPU Systems These types of systems are the most common (and only in the last few years have we begun to see a lot of research on the topic of GPU graph processing). There are several solutions in this category. From hard-wired implementations of graph primitives to several frameworks like Ligra [30], that abstract the programmer from low-level parallelism details, and allow focus on the development of the actual graph primitives. These solutions can take advantage of multi-core architectures and can achieve good performance, however, they are still significantly slower than recent GPU implementations.

Distributed CPU Systems These systems take advantage of several machines with several CPUs, splitting graph computations between them. There is an obvious advantage of scaling the solution to multiple machines and CPUs, however, these types of solutions incur in a high communication cost in order to synchronize information between the different machines and a high monetary cost of acquiring the hardware or renting a cloud infrastructure.

Low-level GPU Implementations These implementations are the best of all the categories. Taking advantage of the GPU's massive parallelism for graph processing allows for very fast solutions, when compared to the other categories. The big disadvantage of this solution is the challenge involved in programming GPUs.

High-level GPU Frameworks These systems are typically slower than low-level GPU implementations, due to the overheads inherent to high-level programming libraries and the lack of graph primitive optimizations that can be achieved when programming low-level solutions. However, the great advantage of this category of solutions is the fact that the programmer is abstracted from the complicated, low-level details of GPU programming and can focus on the actual development of the primitive. Additionally, the overheads of using a framework are usually negligible when considering the advantages.

Considering the focus of this thesis is not in the development and improvement of graph primitives or graph processing libraries, we chose to use a solution that falls in the High-level GPU frameworks category. Like stated in the beginning of this section, using GPUs to accelerate our overall solution is one of our design goals. Thus, we can benefit greatly from using a framework that allows us to focus on the actual implementation of the graph primitives we require (or even use a library that already has it implemented) and still take advantage of the massive speed-up offered by computing these primitives on a GPU.

2.4.2 nvGRAPH

nvGRAPH [24] is a library developed by NVIDIA, comprised of graph primitives with high performance and developed in CUDA. It provides graph construction and manipulation primitives, and a set of graph primitives optimized for the GPU. nvGRAPH views and solves graph problems as linear algebra problems and matrix computations. Currently, nvGRAPH supports three algorithms: page rank, single source shortest path and single source widest path. It is also possible to use nvGRAPH operations to build other graph transversal algorithms. This library falls into the high-level frameworks categories listed above and, although it requires some CUDA programming knowledge, it still abstracts the programmer from low-level details. It allows great control of how the workflow is processed, enabling extraction of subgraphs, execution of operations on graph nodes (vertices or edges) and data reformatting. The results can be downloaded from GPU to host memory and also copied to another location on the GPU. Multiple algorithms can be run while the graph is in the device memory, which allows us to bypass the overheads of loading data into the GPU, if need be.

2.4.3 GunRock

GunRock [38] is a state of the art, high-level GPU graph processing library. It provides an easy way to execute graph primitives on GPUs, abstracting the programmer from low-level GPU details. This falls into the high-level GPU frameworks mentioned in the previous subsection. GunRock provides five primitives, already implemented: breadth-first search, betweenness centrality, single-source shortest path, connected component labelling and page rank. Additionally, it provides the ability to develop new graph primitives with ease, requiring low code usage and little to no GPU and GPU programming knowledge.

GunRock has comparable performance to low-level graph primitives (only differing in the little to no overheads inherent to high-level frameworks) and, to our knowledge, performs better than any current high-level graph libraries available. Also, the authors of [38] recently extended GunRock to support multiple GPUs for the computation [27], which is advantageous for our work, enabling us to better scale our solution by using a GPU cluster.

This library differs from most by being data-centric, focusing on the manipulation of a data structure named *frontier* that contains the vertices or edges that represent the subset of the graph currently participating in the computation. Also, unlike many other libraries it allows the primitives to compute on vertices or edges, instead of only one of them exclusively. It functions solely by bulk-synchronous operations (different steps may have dependencies, but individual operations within a step can be parallelized) that manipulate the frontier: either computing on values within the frontier or computing a new frontier from the current one. Primitives are implemented in three sequential steps that manipulate the frontier in different ways:

Advance This step generates a new frontier by using the neighbours of the current frontier. Due to the possible difference in vertex degrees of each node and the possibility of different nodes sharing neighbours, this step is irregularly parallel.

Filter This step generates a new frontier from the current frontier by choosing a subset of vertices or edges that fall within the specified filters. Due to the possible irregularity of the filters and the nodes they select, this step is also irregularly parallel.

Compute The compute step defines an operation to be executed on all elements (vertices or edges) of the frontier. The specified operation is executed in parallel across all elements selected by the advance step and filtered by the filter step, making compute a regularly parallel step.

GunRock programs consist of the actual problem (containing the graph topology and the algorithm-specific data management interface), functors (which are user-defined computation, applied to the filter and compute steps) and the enactor (which is the entry point for the graph primitive, specifying the computation as a sequence of compute, advance and/or filter kernels). Multiple kernel calls to execute the different steps described above incur in bandwidth intensive intermediate computation steps in order to load data into the GPU and read it back to the CPU, meaning high overhead costs, which may become unbearable if the graph is sufficiently big. In order to obtain better performance, GunRock features kernel fusion, where the condition (filter) and apply (computation) functors are integrated into the advance and filter step kernel calls in order to save memory bandwidth and smoothing out the irregularly parallel operations.

2.4.4 Conclusion

The superior workload mapping and load balancing briefly discussed above, gives GunRock an advantage over other available graph processing GPU frameworks. The performance benchmarks executed by GunRock's authors in [38] show exactly that. It has comparable performance to low-level GPU implementations and better performance than any other high-level GPU framework. nvGRAPH also has great performance, although it is slightly worse than GunRock in most cases. The high programmability, algorithm support and better performance of GunRock makes it a better choice. Since our focus is to use the GPUs' processing power for this task, and be able to focus on the development of graph primitives rather than the intricacies of GPU development, GunRock is the obvious choice.

2.5 Distributed Stream Processing

In order to distribute the workload and take advantage of using a GPU cluster, we can take advantage of a distributed streaming framework. Such a framework allows us to, in a more easy fashion, execute computation in several nodes, while also providing us

with fault tolerance. Frameworks like these usually work by specifying a topology of nodes, with directed edges that specify the data flow. Each node executes some type of computation and passes the data forward according to the specified topology. This would allow us to specify the nodes as explained in section 1.3.1 to perform each task, as well as the data dependencies between nodes.

2.5.1 Apache Storm

Storm [10] is an open source distributed real-time computation system. Storm consumes streams of data and processes them in a specified way, partitioning the stream between different stages, specified in a topology. It is also scalable and fault-tolerant. This makes it very useful for our purposes, due to the fact that we intend to use a GPU cluster (thus taking advantage of Storm's scalability) and that we intend to process streams of data.

A storm cluster contains two types of nodes: master and workers. The master node, called the *Nimbus*, is responsible for distributing work around the cluster (between the worker nodes) and monitoring for failures. The worker nodes listen for work assigned to them by the *Nimbus* node and start or stop processes according to that.

Storms works by specifying a topology, which is a graph of computation. A topology is composed of several nodes spread across several machines. Each node in the topology contains logic for a worker to execute and the edges between nodes specify dependencies and data flow. The worker nodes process a subset of the topology.

Another abstraction in Storm is the stream. Storm provides primitives to handle and process a stream, through *sprouts* and *bolts*. *Sprouts* are a source of streams, while *bolts* consume input streams and process them. We can specify a graph of bolts and sprouts (which are nodes) with the edges between them signifying stream dependencies. This is illustrated in Figure 2.12.

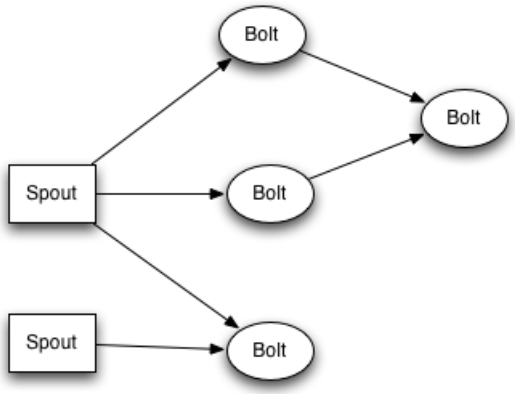

Figure 2.12: *Bolts and Sprouts Topology*

The Sprouts (on the left of the image) are the source of streams, and pass them to the three bolts in the middle of the image. Once these execute the necessary processing, they

pass the streams on to the final bolt.

Storm is simple to use, easy to configure and its abstractions allow us to easily implement a distributed stream processing system. Since Storm works with any programming language, there are no concerns about its compatibility with our system. Also, its stream processing capabilities enable us to easily process streams of images in any way we desire.

2.5.2 Intel Threading Building Blocks

While Storm allows us to take advantage of a cluster of machines horizontally, Intel Threading Building Blocks (TBB) [6], allows us to take advantage of each of the machines vertically. It enables a programmer to easily write C++ parallel programs that take advantage of modern multicore processors. It is widely used and tested, meaning it is very reliable. It includes several algorithms, locks and atomic operations that allow a programmer to abstract from low-level details, worry less about the intricacies of parallel programming and focus more on the design of the solution.

We can take advantage of TBB in order to accelerate our solution in each machine that we use, taking advantage of as much processing power as we can. In modern computers, no part of the processor should be left idle, and TBB allows us to make sure that never happens.

2.5.2.1 TBB Flow Graph

TBB Flow Graph is a TBB interface that enables a programmer to easily write powerful dependency graph and data flow algorithms. It expresses computation in terms of directed graphs, with dependencies between them. Data flows in the direction of the graph edges, and it is possible to have several graph nodes executing in parallel. There are also different types of nodes to help with the construction of the flow graph, such as buffer nodes, split and join nodes, queue nodes, and several others.

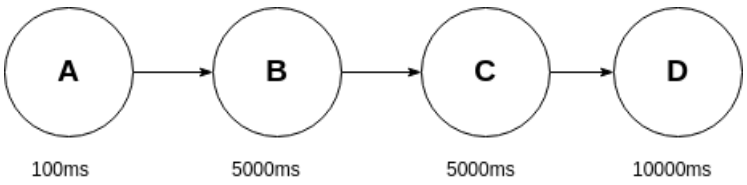

Figure 2.13: *Serial Execution*

Imagine a program composed by four components A, B, C and D. Component A does some initialization while components B and C perform independent calculations on the initial data generated by A. Component D, that depends on the data calculated by B and C, finally computes the final result. A serial execution of this algorithm can be seen in Figure 2.13. As we can see, the components execute sequentially and take a total of 20100ms to execute. However, components B and C only depend on data coming from A, meaning they can execute in parallel. Using TBB flow graph, we could design

something like shown in Figure 2.14. This way, after the computation is executed on A, components B and C execute in parallel and send their results to the final component D, which performs the last computations. Through the use of TBB we can reduce the total execution time to 15100ms. This basic examples illustrates the usefulness of TBB flow graph and shows how we can take advantage of it for our computations because, as shown in Chapter 1, our program can be expressed in terms of a flow graph. Flow graph also has another very useful feature, it allows each node to be parallelized to a certain degree, *e.g.*, we could (depending on concurrency constraints) make each independent node execute in several processor cores simultaneously, speeding up the process even further.

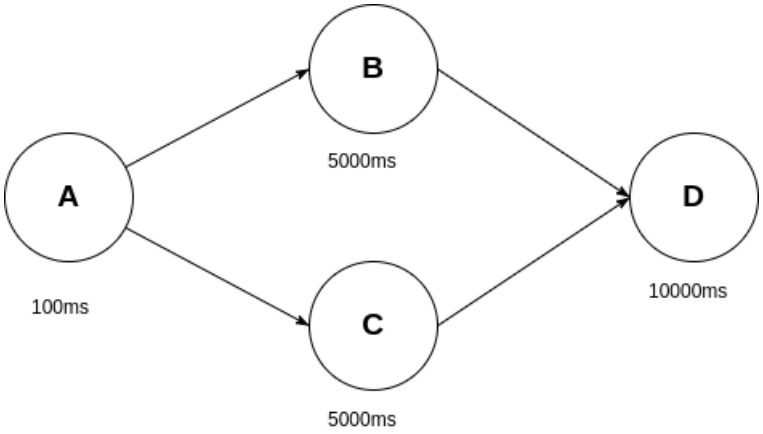

Figure 2.14: *Parallel Execution*

2.6 Multimedia Stream Processing in GPU Clusters

As we explored in this chapter, there is several work on the topic of investigation focusing on GPU image processing (feature detection and extraction) and matching, GPU graph processing (from graph construction to the actual graph processing primitives), and GPU k-nearest neighbour algorithms.

Very recent work by Johnson et al. [17], implements a state of the art, billion-scale image similarity search on the GPU, which is very in line with what our work aims to be. This library constructs a high accuracy k-NN graph on 95 million images in 35 minutes and a graph connecting 1 billion vectors in around 12 hours using hardware very similar to what we intend to use, and is very efficient at searching it. The key difference between our work and the work in [17] is the fact that we plan to use multiple machines equipped with GPUs working in parallel instead of a single machine with multiple GPUs. Our solution aims to scale linearly with the amount of machines that execute it, and it also means that different machines can execute different tasks (such as some machines downloading images, some extracting features and others building the graph). Another important aspect that differs from Johnson's work is the fact that our library, packaged together, not only processes the actual images (ideally extracting any

type and combination of features we desire) but it also constructs any number of graphs we intend (different graphs for different image features, such as the ones extracted from different algorithms) and enables querying on the graphs. It intends to be a full image processing and querying solution. The goal is not requiring images to be preprocessed or pre-downloaded, but downloading and processing them as we construct the graph and having the ability of switching between different feature detection, description and matching algorithms.

The work in [37] also uses GPUs and map-reduce techniques to retrieve near-duplicate videos. Although our focus is not videos, several components of this work are similar to our own: GPU usage, computer vision algorithms, map-reduce techniques and similarity computations.

The work in [15], by researchers at Facebook, uses the power of distributed computing to process videos at a very large scale that is consistent with the requirements of a website of that scale, with more than 8 billion video views per day.

Hartley et al. [14] use a cooperative parallelization approach to implement a large-scale biomedical image analysis application. They also use a cluster of GPUs and multi-core processors.

This last work is also similar to the one in [33], which also uses a cluster of GPUs and multicore processors for the processing of pathology image datasets implementing, similarly to us, an application expressed as a pipeline of different stages, each stage being partitioned into fine-grained operations, to achieving cooperation between GPUs and CPUs, maximizing parallelism.

Chapter 3

Similarity Search

In this Chapter we will explore a high-level view of our solution, the algorithms it uses, the operations it provides and its architecture. This Chapter does not detail the implementation of the system, but what it achieves. In Chapter 4, we will explore the finer implementation details.

3.1 System Description

Similarity Search (SS) was designed to be a distributed, high performance computing system that processes and compares images between each other to output a graph with nodes that represent all the processed images and edges that contain a similarity value between the linked images. It is designed to be modular (meaning we can swap out different components and algorithms with ease), scalable and simple to use.

The graph (or graphs) that this program outputs contain a lot of diverse information regarding the images, which means they can be processed in a number of different ways. A user can use our program to perform certain types of queries (such as searching for an image's n most similar images) or he can take advantage of the graph information to design his own types of queries and operations.

The program is designed to extract features from images using one or several algorithms. Besides this feature extraction process, the program also reads image *metadata* in order to extract the time at which the picture was taken. After the initial image processing (feature and time extraction), the program performs the comparison component, where it compares images against each other in a number of different ways:

- k-NN Brute-force Matching
- Histogram Comparison

- Time Comparison

After this process is complete, the system outputs one graph for each algorithm used, plus one graph for time. An overview of this execution model can be seen in Figure 3.1. As we can observe, the input to the system is an image stream and the output is the graphs. The Source Node is responsible for processing the incoming stream of image URLs and downloading the images. It also extracts some image information such as its name, and stores it for later use. After this initial processing, and after the image is downloaded, it is forwarded to a GPU Load Node, responsible for uploading image information to the graphics processing unit. The processing then goes on to the feature extraction portion of the program. The feature extraction nodes each extract features from the images (using the GPU) using a certain computer vision algorithm. These features are then forwarded to the feature matching nodes. These final nodes calculate image similarity for every pair of images that arrives and construct the graph that our solution outputs.

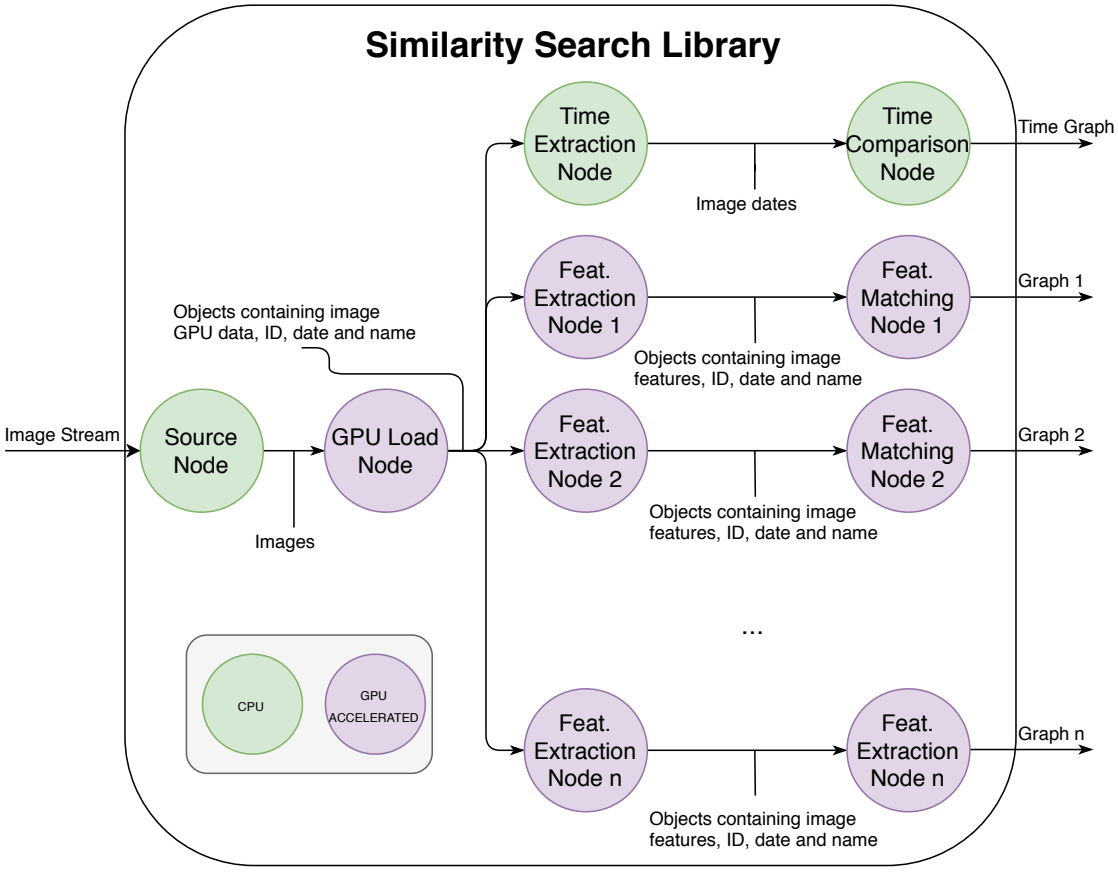

Figure 3.1: *Similarity Search top-level Flow Graph*

3.1.1 Feature Extraction

For the steps of processing images (feature extraction and matching), we made the decision to use OpenCV, due to the fact that its a very complete, powerful library with wide

support and it has a big community involved in its development. Every aspect of image processing (image GPU upload/download, image feature extraction/description and image feature matching) is handled by OpenCV. The four computer vision algorithms our program supports are SURF, HOG, ORB and VGG. The decision to go with these algorithms is due to the fact that they each have their uses, advantages and disadvantages. While these algorithms were chosen, usage of OpenCV means that the addition of more algorithms is simple. Using our program, it is possible to specify which combination of algorithms one wishes to use. For instance, a user may decide to run the program only using the SURF algorithm, or the SURF and ORB algorithms, instead of all simultaneously. However, executing the program with all algorithms simultaneously means the output is very rich and diverse. This means that the output graphs can be used for a number of different types of comparisons (*e.g.* human detection and object detection).

SURF Speed-up Robust Features was chosen due to its speed over its peers. Both feature extraction and matching have great performance. The precision of these features is also at an acceptable level, and we can tell how similar two images are with some degree of confidence.

HOG The decision to use Histogram of Oriented Gradients was due to its precision. These features are very accurate at describing an image and we can obtain great results on image similarity calculation. Out of the three algorithms, it is the one that yields the most accurate results. However, it does have its downfalls. While feature extraction is quick to execute, feature matching is significantly slower that the other algorithms chosen. This impacts performance quite a bit. It is, however, a trade-off for more precise features. Additionally, the memory usage of this algorithm is larger than in the others (due to its features' much higher dimensionality), although some optimizations can be made to reduce the features' size, without sacrificing much of their accuracy.

ORB Oriented FAST and Rotated BRIEF was chosen due to its very low memory usage. When compared to the other algorithms, its impact on memory is much lower. It also has very good performance, and is very close to SURF in that aspect. In regards to precision, however, it is the less precise of the three.

VGG VGG features are very precise but extremely slow to extract. This makes the whole pipeline slower and introduces starvation in the feature matching nodes, which sometimes have to wait for features to be computed and fed to them in order to continue processing. Due to this, and considering OpenCV does not have GPU support for VGG, there was a decision to omit the VGG algorithm in the final application, although the system supports it.

3.1.2 Feature Matching Algorithms

After the feature extraction process, the feature matching nodes have to compare images between each other and output a similarity score. There are several ways to achieve this, but we chose only two: brute-force k-nearest neighbours for SURF and ORB features and histogram comparison for HOG features. OpenCV features several more feature matching methods, which means it is easy to implement more into our program in the future.

Brute-force k-NN As described in the previous chapter, brute-force k-NN performs feature matching by finding, for each feature in the query image, its k closest features. We use $k = 2$ in order to use Lowe's ratio test [21] to filter out good matches. In order to calculate a similarity score, we divide the number of good matches (the ones that pass Lowe's ratio test) by the number of key points in the query image. This measures score well, because it is a ratio of how many of the key points in the query image match the second image. If all the key points match, it means the images are the same. There are a few difference between brute-force k-NN for SURF and for ORB. In the case of feature matching of SURF features, the norm used (*i.e.* the distance measurement to be used when finding matches) is the Euclidean distance, whereas in the case of feature matching of ORB features, we use the Hamming distance. For both algorithms, the ratio used in Lowe's ratio test is 0.7.

Histogram Comparison For HOG features, the process is a little different. Since the feature descriptor that the HOG algorithm outputs is a histogram, we can not compare it using the brute-force k-NN methodology just described. OpenCV provides, however, a handy way to compute a numerical value that measures how well two histograms match. This is all we need to compute a similarity score. This computation can be done in several different metrics, however, the one we chose was *correlation*, which is calculated in the following manner:

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - H_1^{'})(H_2(I) - H_2^{'})}{\sqrt{\sum_I (H_1(I) - H_1^{'})^2 \sum_I (H_2(I) - H_2^{'})^2}}$$

where

$$H_k^{'} = \frac{1}{N} \sum_J H_k(J)$$

and N is the number of histogram bins.

The reason we chose the correlation metric is that the equation outputs a value between 0 and 1, where the higher the value, the closer the match. This represents a simpler value to process, as other histogram comparison metrics output more arbitrary values.

Both comparison methods we just described output a value between 0 and 1 representing how similar the images are. If the value is 1, it means the images compared are the same. For processing purposes that will be explained later, we subtract this score from 1, in order to obtain its complement. This means that if the score is 0 the images are the same and, as it approaches 1, it means the images are more and more different. However, in order to make the score a more round, readable value, we decided to multiply it by 100 and round it to the nearest integer number. This decision is arbitrary and only serves the purpose of providing integer scores between 0 and 100 (0 being the most similar and 100 being the most different). It also serves the purpose of normalizing scores throughout our solution.

3.1.3 Time Comparison

Besides the comparison of image features, our program also compares image dates, *i.e.* the date that the image was taken in. The comparison is simple, we calculate the absolute value of the subtraction of both dates. In our solution, the date difference is presented in seconds. As seen previously, one of the outputs of our solution is a time graph. It is like the other graphs, except the edges contains time difference values instead of image similarity scores. All the other graphs also have access to the image date values. This way, it is possible to design a query that factors in the time difference with the visual similarity of the images. A user can also take advantage of the time graph to extract a multitude of information regarding the images in time (*e.g.* the closest images taken in time).

3.1.4 Input: Image Stream

The input to our system is an image stream containing URLs. The source node reads the image URLs and is responsible for downloading the images to be later processed. The image stream may also contain different information regarding the images, such as their date. It is up to the source node to process this stream and extract whichever information is relevant to the user's desires. In our case, we are interested in image dates as well as the actual images, so our source node needs to process the stream in a way that not only downloads the images themselves, but extracts their date.

It is easy to change the stream source to a different one, following the modular design goal of our solution. One merely needs to swap out the source node with another that processes the stream in its format, generates image URLs, downloads the images and extracts the relevant information.

3.1.5 Output: SS Graphs

The output of the Similarity Search system is a collection of graphs (one per computer vision algorithm used), each containing all the information about the images that were processed. It is composed of nodes that contain image identifiers and image features as

well as undirected edges that contain the image scores. Our graph data structure stores the following information:

- Nodes, represented by a map that stores image features indexed by integer image identifiers.
- Edges represented by a vector containing matrix coordinate objects (triplets that store two image identifiers and a numerical value, *i.e.* the similarity score).
- Image dates, represented by a map that contains the image dates as values and uses the image identifiers as keys.
- Image objects (which we call SS Images) comprised of diverse information regarding images, such as their identifier, their file name, their date and their features. This image object is also serializable and contains several functions that enables us to read from and write to files. Their serialization also enables us to easily share these objects through the network, which is a major necessity as we will see later on.

The graph also contains diverse functions that allow users to do a multitude of operations such as executing a query, retrieving nodes, edges, image dates and among others. Besides the graph built-in operations, users can take advantage of it to design any kind of operations they wish. It can also generate a Gunrock-compatible graph, meaning any of Gunrock's primitives can be executed on this graph.

3.1.6 Operations

There are a few essential operations that our system presents. In this subsection, we will go over what they are and what they do. We will save the implementation details and other intricacies for future sections. The operations are as follows:

Feature Extraction and Matching This is the main, most important operation of the system. By supplying this operation with images, it performs the necessary computations in order to output the SS Graph previously discussed. It extracts features and matches them using the algorithms specified in Subsection 3.1.1 and Subsection 3.1.2. A user can also choose the combination of algorithms he desires. There are also a few other options. If the user chooses to do so, this operation will save image information in a compressed text file. This maps image identifiers to image names (which are derived from their URLs) and allows users to browse and view that information, should they need to do so. It is also possible to save image features to another compressed text file. This removes the need to download and process an image again, should further comparison of that image need to be made. Finally, there is also the option to save image scores as a symmetric matrix in a text file. This allows manual browsing of image scores, or the loading of the scores to perform a number of more queries. These file input/output operations allow some type of

persistence in our program, and allow users to keep their data accessible at all times, even if the program is not running. Additionally, it allows work to pause/resume at any time.

Queries Another important feature of our program is the ability to perform similarity queries. The queries are executed on the SS Graphs outputted by the feature extraction and matching operation. It allows users to provide either one or several image identifiers (of already processed images) or an image URL (which SS then processes and indexes in the graph). SS then browses the Graph, using the GPU, and investigates which images are more similar to the ones provided. It is also possible to modify the query to retrieve the n most similar images, instead of just the most similar. This is the base query supported by our system and, although we assume more could be implemented in the future, the power of the SS graph that our solution outputs can be processed in a number of different ways. The graph also has an operation to write its information to a matrix-market format file, which is the input to Gunrock primitives. This way, it is possible process image information with a number of Gunrock primitives, such as betweenness centrality, breadth-first search, single-source shortest path, page-rank, and many others. Users can easily design their custom queries using not only the SS Graph datastructure, but the ability to convert it to a Gunrock-compatible graph.

It is important to note that the components that make up these operations are heavily customizable. For instance, the images that the feature extraction and matching process take as input can come from any stream source. They can even come from the features compressed text file (bypassing the feature extraction process). We can also load both the features file and the scores file, which means we can bypass the whole feature extraction and matching process, and move on straight to queries or to the processing of new images. The matching process can also be tweaked to output a different type of score, or a score that factors in more than image similarity (*e.g.* image geolocation and image date). The possibilities are endless, and the modular design of our system allows changes like these to be easily implemented. It is also possible to change the algorithms used in the feature extraction process, or even the techniques used in the feature matching step. The modularity of our solution is further detailed and evidenced in Section 4.4.

3.2 System Architecture

In order to distribute our solution, it is necessary to physically distribute some of its components. In order to achieve maximum performance, and considering that the most computationally expensive part of our program is the feature matching component, we separate feature matching from feature extraction. Considering this fact, there are two roles each machine or process can assume: *feature extractor* or *feature matcher*. As the names suggest, *feature extractor* is responsible for extracting features. It also processes

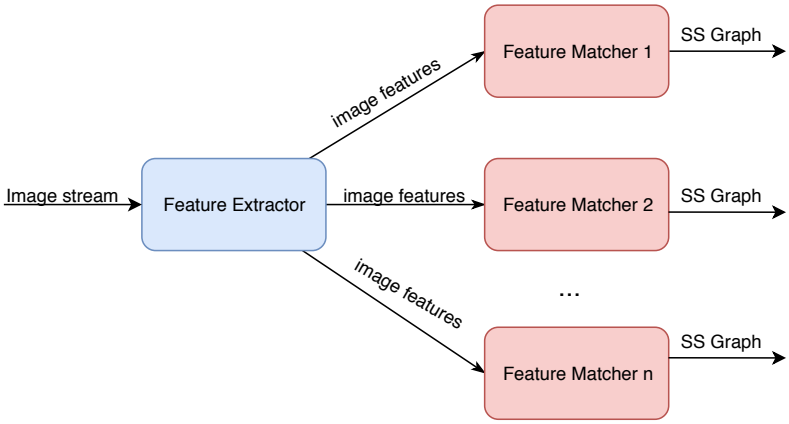

Figure 3.2: *Similarity Search Architecture*

the stream and downloads images. It then forwards the feature descriptors to the *feature matcher*, which performs the similarity calculations and constructs the SS graph. There can be more than one of each type of roles, each extracting features and processing them in a certain algorithm. It is also possible to have the *feature extractor* role extract features from more than one different algorithm. In our case, we have one *feature extractor*, responsible for downloading all images, extracting feature descriptors from three algorithms, and forwarding them to three *feature matchers*, each responsible for the similarity calculation for one of the types of features, but the possibilities are endless. The architecture we discussed can be visualized in Figure 3.2.

3.2.1 Feature Extractor Role

The flow of the *feature extractor* can be seen in Figure 3.3. Its responsibilities are, first of all, processing the image stream (whichever it may be) to extract image URLs and, subsequently, downloading the images. The images are then loaded to the GPU, and shared with the other nodes (time and feature extraction). If enabled, the GPU load node will also be responsible for forwarding the image identifier-name pairs, in order to be saved to a compressed text file.

The time extraction node reads the image metadata and extracts its date. The date is forwarded to the time comparison node that compares all dates received and calculates scores (time difference in seconds), outputting the time graph. The reason why time is compared here, as opposed to comparing it in the *feature matcher*, is to keep the *feature matcher* specialized in its task, and keep as much of its processor time (both CPU and GPU) dedicated to matching features rather than comparing dates. Since the comparison of dates is significantly faster (a mere subtraction operation) than the comparison of image features, we can afford to keep it in the *feature extractor* role. Additionally (and as we will see in further detail in Chapter 4), the time comparison step is executed in parallel with the feature extraction steps, meaning it has little to no impact on overall performance. The addition of this step here slows neither the *feature extractor* role nor

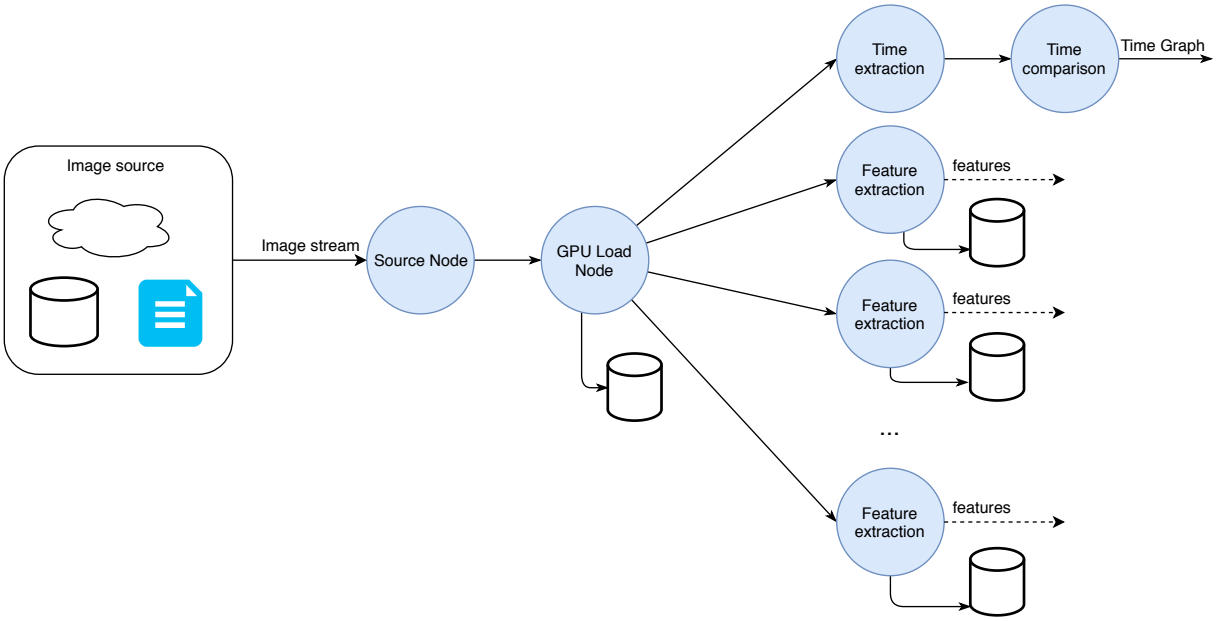

Figure 3.3: *Feature Extractor Flow*

the *feature matcher* role.

Finally, the feature extraction nodes compute feature descriptors from the images received using one of the algorithms our system supports and forwards the features along to the next role: the *feature matcher*. If enabled, they also write the image features to a compressed text file, in order to save them for later use.

3.2.2 Feature Matcher Role

The flow of the *feature matcher* role can be observed in Figure 3.4. This role receives the image features of one of the types of algorithms our program supports. As this role receives features, it compares them with the ones it already has, calculating a similarity score, which it then forwards the the *score indexing node*. This last node is responsible for indexing the incoming scores into the graph, which it then outputs when the execution is complete. As the score indexing node outputs the graph, it may also (if the option is enabled) save the image scores for later browsing or processing.

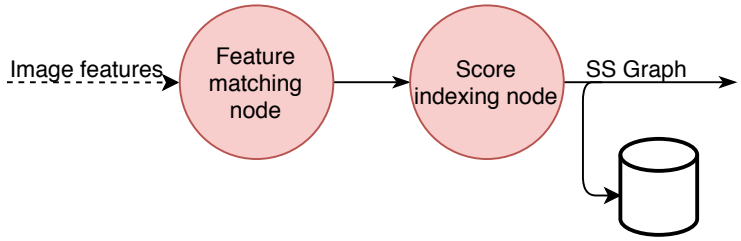

Figure 3.4: *Feature Matcher Flow*

Chapter 4

System Implementation

All of the functionality described in the previous chapter has a lot of details hidden behind it. In this chapter, we describe the implementation of the features previously explained in greater detail, explaining the technologies used, the logic behind each component and how they work.

Figure 4.1 details the complete flowgraph of our solution and their mapping to each cluster node. As can be seen, there are a lot of nodes and data flow edges, both local and through the network. This chapter will provide the reader with knowledge of what each node does and how it does it.

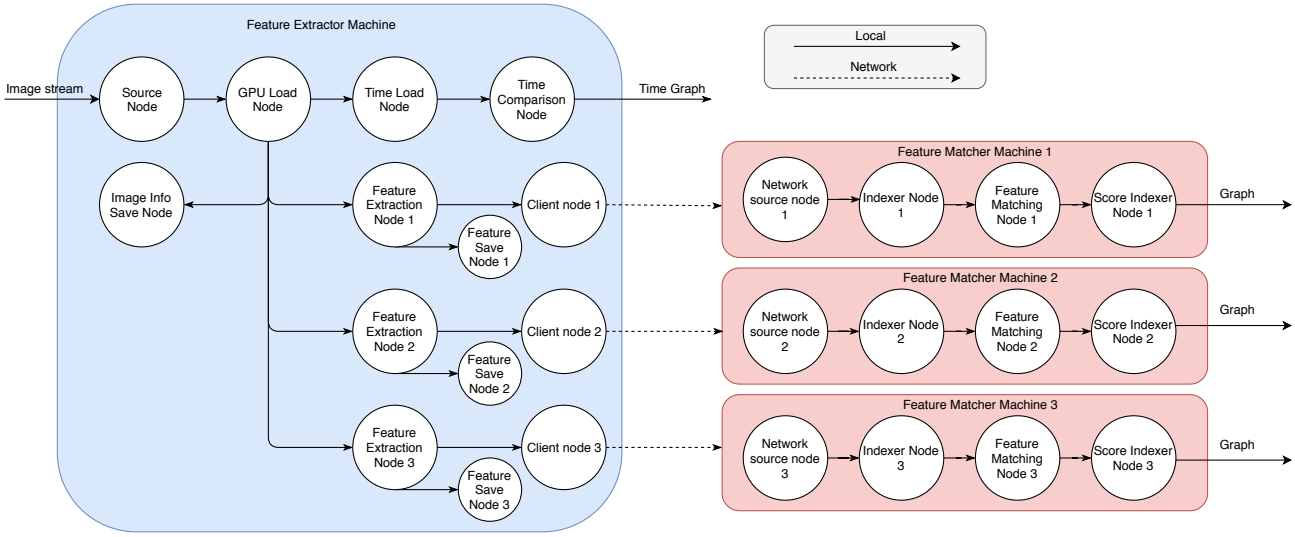

Figure 4.1: *Complete flowgraph*

4.1 Technologies used

The system was written using *C++*, because it enables us to write optimized, fast code. Additionally, a lot of the libraries used have a *C++* implementation and this enables us to take advantage of them. We used several libraries and programs to develop our solution and in this section we will go over them and why they were chosen.

4.1.1 OpenCV

As discussed in the previous chapter, in order to process images, we use the OpenCV library. The choice of this library is due to the fact that it is arguably the most complete computer vision library available, with very complete documentation and support. It is written in *C++* (with Java and Python interfaces) and designed for real-time applications. It takes advantage of multi-core architectures and it is written to be as optimized as possible. It can also take advantage of hardware acceleration (such as the use of GPUs) and it has both OpenCL and CUDA support. A lot of the computer vision algorithms that OpenCV supports have a CUDA implementation packaged and ready to use, which helps our work a lot and allows us to focus on our solution rather than in the implementation of computer vision algorithms for the GPU.

We take advantage of OpenCV in order to extract feature from images, using some of its many implementations of computer vision algorithms (such as the ones discussed in Section 2.2). We also use OpenCV's feature matching capabilities (*e.g.* brute-force k-NN) in order to obtain matches between images and enable us to later calculate a similarity score between them. Besides that, we use OpenCV's FileStorage to enable the easy writing and reading of information to and from files, should it be necessary. As we will discuss further below, our system enables the option of saving image information (such as identifiers, names and features) for later use. Besides the usage of these functionalities, OpenCV also handles the reading of images and their respective conversion to a matrix format as well as their upload to the GPU, plus any additional image pre-processing that needs to be done before the extraction of features, such as image conversion to a different scale or color palette.

4.1.2 Threading Building Blocks

Intel TBB flow graph [16] is used in order to express our computation in terms of a flow graph. As Figure 4.1 implies, the computation we execute can be expressed as a flow graph, with nodes that indicate computations and edges between nodes that express the data flow between them. TBB flow graph helps us to express such a computation, designing a graph with different data dependencies between nodes. It also helps us to parallelize our solution, since each independent node can be run using as many threads as a user wishes.

TBB flow graph provides several types of nodes, each with different functionalities and use cases. Here, we showcase the ones we take advantage of:

Source Node The source node is the node where all flow graphs begin. It has no predecessors. Also, it is always serial so it can never be executed concurrently. Its important then, to make sure that its body is not a bottleneck and does not starve its successors. The source node works by evaluating a condition which, if true, will keep producing more data. If no successors accept the message it produces, it will be kept in a buffer until another successor is added or requests it.

Function Node This node receives an input, applies its body to it, and outputs the data to its successors. It is the main computation node, as it receives data, performs calculations and broadcasts the output. The (maximum) concurrency of this node can be specified in its constructor. If one sets the concurrency to *tbb::flow::serial*, its body will never execute concurrently, while if its set to *tbb::flow::unlimited*, there will be no concurrency limit. When set to *tbb::flow::unlimited*, it is important to note that as soon as a message arrives to the function node, a task will be spawned to process it, but this does not mean, however, that a thread will be created. Tasks can only spawn threads from the available library's thread pool. This means there will be no excess of threads, resulting in hindered performance. One can also set the concurrency level to a specified number such as 8 which means that, at most, 8 tasks will be spawned.

We also iterate upon TBB flow graph and, with the help of the Marrow library in [23] and [9], add additional functionality. We create two additional nodes, that allow us to spread this flow graph through the network, and not be limited to one machine. This is the functionality that Storm offers. We decided to opt with this solution instead of storm, due to the difficulty that lies in implementing a solution that has to constantly load information to and from the GPU, as well as execute algorithms on it. It would introduce unnecessary overheads to design a solution that uses Storm, as a lot of information would have to be loaded to the Storm topology and then fetched from it. The solution we designed turns out to be much simpler to implement, easier to use and very efficient at its task. There was no need to introduce additional complexity to our solution. Its objective was to design two nodes that enable communication between machines, and abstract TBB from the fact that the two nodes connected are, in fact, residing on different machines. As far as TBB is aware, the flow graph on the first machine ends on the node responsible for sending the information to the remote machine. On the second machine, TBB knows nothing about where the data is coming from, it is merely reading it from a node that parses information from the network, just like a typical source node would. Essentially, we split a flow graph in two, distribute it between two different machines and introduce a way to communicate necessary data between them. In order to support this functionality, we introduce two new nodes:

Network Source Node The network source node has a predecessor that resides on another machine. It serves as a typical TBB flow graph source node in order to start a new flow graph in a different machine. It feeds on information sent to it by the client node (described below) through the network. In order to create a network source node, the user needs to specify the port in which the server will run. Afterwards, the node will listen on that port for incoming data and, as soon as it arrives, it feeds it to its successors, just like a source node would. The dashed lines in Figure 4.1 illustrate this communicate between two different machines using a network source node and a client node.

Client Node The client node has a predecessor (typically a normal TBB flow graph function node) that sends information to it. It then sends that data, through the network, to a network source node. The successor of this node is always a network source node residing on a different machine. A user need only specify the hostname and port in which the network source node is running, and it will take care of the rest, sending data along, and successfully enabling communication between machines through the use of flow graphs.

4.1.3 Google Protocol Buffers

Google's protocol buffers [13] allow the serialization of datastructures in an efficient, fast and language independent way. In order to create serialized data, a user defines how he wants the data to be structured in a *.proto* file, specifying information about the datastructure such as the data fields, their type and whether or not they are mandatory fields. Afterwards, the protocol buffers compiler reads the *.proto* file and generates code that allows the easy writing and reading of data. It enables serialization in a number of different ways such as serializing to a string, to bytes, to a file descriptor and several others. It also enables the easy de-serialization of data, in order to reverse the process.

We take advantage of protocol buffers in order to serialize our datastructures, allowing us to send them through the network with ease (this is the data that flows in the dashed lines shown in Figure 4.1, between the client and network source nodes). The advantage of protocol buffers is that they support a multitude of languages, including *C++* and that they are incredibly fast and light, meaning we minimize the overheads of serializing our data before sending it and de-serializing it after receiving it, avoiding potential performance issues from doing so.

4.1.4 Gunrock

The Gunrock graph processing library is designed specifically for execution on the GPU. Graph processing is a very computationally expensive part of our program, since the graphs contain potentially thousands of nodes and hundreds of millions of edges. We take advantage of the graph primitives offered by Gunrock in order to perform computations

on our outputted graphs, namely the computation of queries. In order to perform a query in our program, we use Gunrock's implementation of the Single-source shortest path (Dijkstra's) algorithm. This algorithm, given a source node, finds the shortest path to all other nodes in the graph. We can then iterate this list of shortest paths to all nodes to find the one with the lowest score (which means the highest image similarity) and return it.

4.1.5 cURL

cURL [7] is a multi-protocol file transfer library and command in unix systems. It provides several tools to send/receive files to/from a remote server. We use this library to download images from the image stream supplied image URLs and save them to the machines, to be used later by the system for image processing.

4.1.6 Boost

The Boost [2] *C++* libraries are, essentially, an extension of the *C++* language. They contain a multitude of datastructures, support for linear algebra problems, multithreading tools, stream processing tools, and many more. In our case, we use the Boost libraries for one task: the conversion of the compressed YFCC100M dataset text file into a stream.

Boost's bzip2 filters allow the compression or decompression of files. We use a decompression filter to process the YFCC100M text file and be able to read its information as a stream. This way, we do not require the full decompression of the file before being able to read it. This helps with performance and allows the processing of the file in order to retrieve image URLs to be downloaded later.

4.2 The SS Image Datastructure

The SS Image is a datastructure that we created in order to facilitate the data flow of our program. It contains information regarding an image, it is serializable (through the use of Google's Protocol Buffers) and easy to read/write from/to a file (using OpenCV's FileStorage). It contains the following data:

Image identifier is a unique integer identifier within our library, with the purpose of differentiating images.

Image name is a string with the name of the image (typically the latest part of its URL), for the easy identification of an image.

Image date contains a string that represents the date in which the image was taken, presented in the format YYYY-MM-DD HH:MM:SS.F.

Image data is a GPU stored matrix representation of the actual image or the image features.

Table 4.1: SS Image API

ss_image()	Empty constructor for the class.
ss_image(**int&** image_id, **string&** image_name, **string&** image_date, **GpuMat&** data)	Constructor
private ss_image(**void*** serialized, **size_t** size)	Private constructor from serialized bytes
private ss_image(**string** serialized)	Private constructor from serialized string
private ss_image(**similarity_search::ss_image&** s)	Private constructor from protobuf ss_image object
void write(**FileStorage&** fs)	Write to file using OpenCV's FileStorage
void read(**const FileNode&** node)	Read from file using OpenCV's FileStorage
similarity_search::ss_image serialize()	Serialize to protobuf object. Returns the protocol buffer object.
string serializeToString()	Serializes the object to a string.
static ss_image parseFromStream(**int** fd)	Parses a stream, constructs the ss_image object and returns it.
static ss_image parseFromBuffer(**void*** buf, **size_t** size)	Parses a buffer, constructs the ss_image object and returns it.
static ss_image parseFromString(**string** str)	Parses a string, constructs the ss_image object and returns it.

The SS Image API can be seen in Table 4.1. It contains diverse methods for serializing or de-serializing the object, as well as methods to read from or write to a file. It also contains several constructors to meet diverse requirements.

4.3 The SS Graph Datastructure

The SS Graph datastructure is the output of our solution. This graph indexes all images as nodes and the scores between each pair of images as edges. It enables the execution of queries and can be processed in a number of different ways, such as converting it to a Gunrock-friendly format and executing a Gunrock primitive on it. As we saw in Chapter 3, more specifically in Subsection 3.1.5, the SS Graph contains the following data:

- Graph Edges
- Graph Nodes
- Image Dates
- SS Image Objects

The API for the SS Graph datastructure can be seen in Table 4.2. It contains diverse methods to add or retrieve information to/from the graph, methods to perform queries and even to convert the graph to a file in the matrix-market format [32], which we will explore next.

4.3.1 Matrix-market coordinate format

The matrix-market coordinate format is a standard for representing sparse matrices, and is used by Gunrock in order to represent the graphs that it loads to the GPU and later

processes. This format is simple and contains, in the second line, three integer values specifying the amount of rows, columns and non-zero entries. Each line after that represents a matrix entry, containing the row number, column number and the respective value. We use it to represent scores between images, where each line contains the two image identifiers and their respective score. It also has some metadata fields that specify details regarding the matrix, such as the data types it contains and whether or not the matrix is symmetric (which is always our case, because the score between image A and B, for instance, is the same as the score between image B and A). Specifying the matrix as symmetric reduces the amount of information written to the matrix-market file by half. An excerpt of an example matrix-market format file, with 200 images and 19900 edges can be found below:

```
%%MatrixMarket matrix coordinate integer symmetric
200 200 19900
2 1 98
4 3 97
4 2 97
4 1 99
3 2 99
3 1 99
5 4 98
5 3 99
...
```

The first two lines contain information regarding the matrix. The first line specifies that its a symmetric matrix, represented in the matrix-market coordinate format with integer data values. The second lines specifies, respectively, the number of rows, columns and edges (non-zero values) that the matrix contains. Each line after that specifies two numbers (row and column) and the value at that position. This means, for instance, that at matrix position *(2,1)* there is a value of *98*. In other words, it means that the score between the images with the identifiers *2* and *1* is *98*. Gunrock can parse this file, convert it to a GPU graph format it can use and then execute its primitives with it.

4.4 Image Processing Workflow

In this section we follow the path the images take, from being downloaded to being compared against others and finally indexed on the graph. Following Figure 4.1 helps with this process.

Table 4.2: SS Graph API

Graph()	Empty constructor for the class.
void addEdge(**int** node1, **int** node2, **unsigned int** value);	Adds an edge to the graph.
void addNode(**int** imageId, **string** imageName, **const cv::cuda::GpuMat&** imageFeatures);	Adds a node to the graph.
void addImageDate(**int** imageId, **string** time);	Adds the image date to the graph.
void addImage(**int** imageId, **ss_image** image);	Adds an SS Image object to the graph.
void printCsrFormat();	Prints the graph in the comparse sparse matrix format (for debugging purposes).
int getNumEdges();	Returns the number of edges in the graph.
int getNumNodes();	Returns the number of nodes in the graph.
int getNumTimes();	Returns the number of image dates in the graph.
void writeMMFile(**const char*** file_name);	Writes the graph to a matrix market format file. This file is used by Gunrock as input to its primitives. This format only writes the right triangular portion of the matrix (as the score matrix is symmetric).
tuple<int, int> getMostSimilarImageMM (**const char** *matrix_format_graph, **int** src_image);	Perform a similarity query using a matrix market file. Returns the most similar image to src_image and the similarity score.
tuple<int, int> getMostSimilarImage (**int** src_image, **bool** quiet = false);	Performs a query. Returns the most similar image to src_image and the similarity score.
vector< tuple<int, int» getMostSimilarImage (**vector<int>** srcs, **bool** quiet = false);	Performs a query on several source images. Returns the most similar image to each source image present in srcs and their similarity score.
map<int, tuple<string, cv::cuda::GpuMat>>& getNodes();	Returns all nodes in the graph.
map<int, ss_image>& getImages();	Returns all SS Image objects in the graph.
map<int, tm>& getImageTimes();	Returns all time objects in the graph.
tm& getImageTime(**int** imageId);	Returns imageId's time object.
string getImageName(**int** imageId);	Returns the image name for the provided image ID.

4.4.1 Source Node

The first step in the images' path is the source node. This node is responsible for parsing the image stream and retrieving image URLs, in order to download the images so they can be processed. It can process any stream necessary, as long as it downloads images and passes them on to the next node. In our case, however, it uses Boost's bzip2 decompressor filter to read from the YFCC100M dataset file, and generate a stream. This stream is processed while there are images to process (in the dataset) or until a user-specified number of images is reached.

The decompressor filter reads from the compressed file, line by line. Each line contains information pertaining to a media file, such as its url, its name, the date it was uploaded and taken in, the geolocation of where it was taken and a multitude more information. The data we retrieve is, for each image, its URL and date. It is important to be aware that this dataset also contains videos so, in order to deal with that, we also retrieve an additional field that states whether or not the current media file we are reading is a video. If that is the case, we ignore it and move on to the next line in the stream, as we only wish to process images.

An image name is constructed from the image URL (corresponding to string after the last / in the URL). This is the file name that is given to the image. After this, the source node checks whether or not this image was previously downloaded and is present in the system's image directory. If not, it then uses cURL in order to download and save the image. A verification is made after the download to check if the downloaded image is

valid, *i.e.* if its data is not corrupted and was correctly written. In case the verification does not pass, the image is deleted from the file system and a new one is downloaded. The image is also given a unique identifier (an integer, incremented each time an image is downloaded). Finally, the source node constructs an SS Image object, using the extracted values (and using a temporary, empty matrix for the image data field), which it then forwards along to its successor, the GPU Load Node. This node is implemented using the TBB source node.

4.4.2 GPU Load Node

This node has a simple task: to take the image file, read it and upload its content to the GPU. It receives the SS image from its predecessor (the Source Node) and extracts the image name in order to construct the image file path. Using this file path, it calls an OpenCV function that reads the image into a matrix. A final function is called, passing the image matrix as a parameter, in order to upload it from the host device to the GPU. The address of the GPU-stored matrix is then returned and stored in the SS Image object. The updated SS Image object is then forwaded to the GPU Load Node's successors: the Time Load Node, the Image Info Save Node and the Feature Extraction Nodes. Note that the GPU Load Node passes one message to each of its successors, meaning they are all called simultaneously and executed concurrently. This node is implemented using the TBB function node and is executed serially, meaning only one image is loaded to the GPU at a time, in order to avoid possible memory conflicts in the GPU. This does not introduce any type of bottlenecks, because the GPU load node is extremely fast and can keep up with the demands of its slower successors. Executing it serially does not starve its successor nodes and it is merely a safety precaution.

At this point, the SS Image object contains the image identifier, the image name, the image date and the image matrix information and it is forwarded to the GPU load node's successors.

4.4.3 Image Info Save Node

This nodes uses OpenCV's FileStorage library to save image information to a compressed text file. The file this node saves serves the purpose of making image name information available to the user. It is a mere assistance to the user and can be enabled or disabled according to the user's desire. It saves image identifier and name pairs, associating each image identifier with the corresponding image name. Because image scores can be saved in a different text file, and they only contain image IDs, this file allows users to verify which image name belongs to which ID. This node has no successor and executes serially, due to the fact that it writes information to a file (a task that would not succeed well if it were executed concurrently).

4.4.4 Time Load Node

This node reads the image date information from the received SS Image, parses it and converts it to the *C++ tm* struct. This struct contains a calendar date and time broken down into its separate components (year, month, day, hour, minute, second and weekday). It then stores the struct object in the SS Graph, for later comparison or consultation. This node is executed serially, as it involves the manipulation of a datastructure (the SS Graph). The successor to this node is the time comparison node.

4.4.5 Time Comparison Node

This is the final node that processes time. Each time it receives an SS Image, it indexes it on the graph (as graph nodes). It also iterates all the different image dates it contains and compares it with the latest one received, outputting the date difference between each pair of compared dates, in seconds. After each comparison, it indexes the comparison on the graph (as graph edges). In order to visualize how this node works, we present the following example, for the time comparison of four images:

1. The time comparison node receives its first SS Image with ID 0 and indexes it in the SS Graph as a node. Since there are no other images present in the Graph, no comparisons are made.

2. The second image arrives with ID 1, which the time comparison node also indexes on the SS Graph (that now has two nodes). Now, it takes the image date of the image it just received (ID 1) and compares it with all the image dates it already has (in the current state, the image date for ID 0). Since the two images are 1000 seconds apart, it adds an edge between the SS Graph nodes 0 and 1 with a value of 1000.

3. The third image, with ID 2 is now processed and indexed to the graph, like the last two. The time comparison node now compares the date of image 2 with the other dates it possesses, image 1 and image 0, outputting values of 500 and 9000 seconds, respectively. Finally, these edge values are added to the SS Graph.

4. The last image now arrives, with ID 3. The process is the same as in the last step, so the date for image 3 is compared with the dates for images 2, 1 and 0, outputting the values 50000, 100000 and 9800, respectively. These final edge values are added to the graph.

A visualization of the output SS Graph can be observed in Figure 4.2.

This node is executed serially, due to its manipulation of the SS Graph datastructure, in order to avoid possible race conditions. Since the comparison of dates is a simple subtraction operation, the serial execution of this node does not impact performance. If we consider the processing of very large amounts of images, however, we may start to notice an impact on performance. To cover this issue, we implemented a version of the

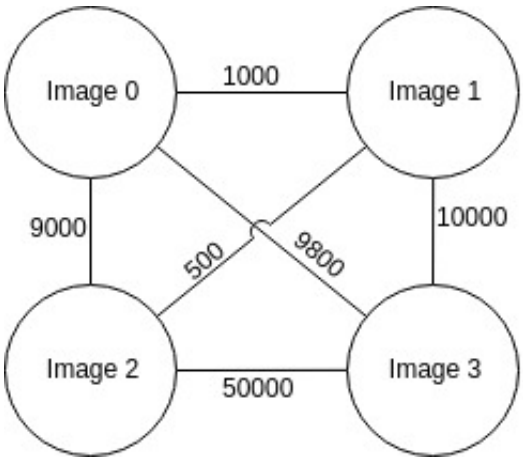

Figure 4.2: *Example time SS Graph*

time comparison node that optimizes this step by using fine-grained parallelism. It has two nodes (one before and one after the time comparison node) that each execute serially, but concurrently in relation to each other and the time comparison node. The node before the time comparison node (time indexer node) handles graph node indexation, and the graph after (time score indexer node) handles graph edge indexation. Since both these nodes are manipulating different parts of the SS Graph, they can execute concurrently in relation to each other, and it is guaranteed that there will not be any race conditions. This way, we can alter the time comparison node to execute concurrently, and modify it so it only calculate scores, that it then forwards to the the last indexer node. This is very similar to the approach we use in the Indexer Node (Subsection 4.4.8), the Feature Matching Node (Subsection 4.4.9) and the Score Indexer Node (Subsection 4.4.10), and allows us to achieve fine-grained parallellism when processing image dates.

This node does not have successors, unless it is executed in its concurrent version. In that case, its successor is the time score indexer node. We omit the details of these extra nodes (time indexer and time score indexer) due to their similarity to the Indexer Node and the Score Indexer Node, explaned in Subsection 4.4.8 and Subsection 4.4.10, respectively.

4.4.6 Feature Extraction Node

The feature extraction node is, as the name suggests, responsible for extracting features from each incoming image. Its predecessor is the GPU Load Node, so each message it receives contains an SS Image object. The feature extraction node extracts the image data (GPU-stored matrix) and then, using OpenCV, extracts features from the image using the specified algorithm. The image features are always outputted in GPU-stored matrix format. After the computation of the features, this node modifies the SS Image object, replacing the current image data it has (the GPU-stored matrix description of the image) with the recently-calculated image features (also in GPU matrix format).

This node is executed concurrently. As we have seen previously, TBB flow graph manages a pool of threads. This means that each time a thread is available, TBB will assign it to execute the body of the feature extraction node. This is advantageous, as feature extraction is typically a much more computationally intensive task the the nodes we visited so far. The successor of this node is a client node, responsible for serializing the SS Image object and forwarding it, through the network, to the respective network source node (that feeds the Indexer Nodes in the feature matching machines). This communication of features between client and network source nodes is, as previously explained, represented by the dashed lines in Figure 4.1. The feature extraction node also has another successor, the feature save node, which we explain subsequently.

4.4.7 Feature Save Node

This node, like the Image Info Save Node, uses OpenCV's FileStorage library to save the image features to a compressed text file. It uses the *write* function described in Table 4.1 in order to serialize and write the SS Image information to a text file. For each SS Image object it receives, it writes the image identifier, name, date and features to a file. This is our system's way of ensuring some type of persistence, and allows loading image features later (in case they need to be compared with new images) without the need to extract them again.

Like the Image Info Save Node, this node can be enabled or disabled as per the user's requirements, executes serially and has no successors.

4.4.8 Indexer Node

The Indexer Node serves the purpose of indexing incoming SS Images in a SS Graph. Its predecessor is usually the network source node, as this node may be executed on a different machine from the feature extraction node. If not, its predecessor is the feature extraction node.

The network source node de-serializes and reconstructs the SS Image object it received through the network, and forwards it to this node. The Indexer Node then adds each image it receives to the graph, as a graph node. It also adds the image date information to the graph, along with the actual SS Image object. Besides this, it also has another important task. The Indexer Node maintains a vector of every node it receives. This vector is global for every indexer node (a member of the Indexer Node class). Each time the Indexer Node adds an image object to this vector, it sends it to its successor: the Feature Matching Node, who processes it.

The reason we do this whole process on a different node rather than in the Feature Matching Node is due to race conditions, as we want the Feature Matching Node to execute concurrently, but the concurrent manipulation of the graph datastructure would introduce several problems. Due to this fact, the Indexer Node manipulates the graph nodes and executes serially.

4.4.9 Feature Matching Node

At the moment, the Feature Matching Node is the most computationally intensive step of our whole solution. In the current implementation, it compares every single image it receives against every other image. This means that for n images, there are ${}^{n}C_{2}$ image comparisons. It is important to note that this step is always slower than all its predecessors, which means that it is always the one that slows down the whole solution. Also, this node will never starve and is constantly performing computations, even with the serialization, network communication and de-serialization of each SS Image Object. These are all very important aspects to ensure, as any type of starvation of this node would introduce major performance hindrances.

This node receives the vector of SS Image objects from its predecessor: the Indexer Node. Similar to what happens in the Time Comparison Node, if the incoming vector has two or more images in it, it will perform feature matching between the last image in the vector and all other images the vector contains. Each time a score between two images is calculated, the Feature Matching nodes places the two image identifiers and their score in a tuple and then indexes that tuple in a vector. Once all images in the current image vector have been processed, it forwards the score vector to its successor: the Score Indexer Node.

4.4.10 Score Indexer Node

The Score Indexer Node, like the Indexer Node, has the task of manipulating the graph datastructure. However, instead of adding nodes to the graph, it adds edges. It receives a score vector that contains tuples, each containing two image identifiers and a score. It iterates this vector and adds each element to the graph, as an edge. This step completes the construction of the graph. This node executes serially, due to its manipulation of the graph datastructure.

This node, paired with the Indexer Node and the concurrent execution of the feature matching node, enable us to achieve fine-grained parallellism in the feature matching step. This is because the indexer node and the score indexer node are concurrently (between each other, but serially in relation to themselves) updating the SS Graph datastructure. However, in order to avoid data races, they are updating different parts of the graph. The indexer node updates the graph nodes, while the score indexer node updates the graph edges. While all of this is happening, the feature matcher node is constantly, calculating image scores in a parallel fashion. With the introduction of this node (and the indexer node), it becomes possible to execute the feature matcher node concurrently, as it does not have to manipulate a global, shared datastructure.

4.5 Query Workflow

A query is executed on the SS Graph datastructure. By calling the appropriate method (one of the getMostSimilarImage functions in Table 4.2), the system will compute the query by executing Gunrock's single-source shortest path (SSSP) primitive. This primitive works by finding the shortest path (with the lowest score) from a certain source node to all nodes that are connected to it by edges, using Dijkstra's algorithm. The first step in this process is the initialization of the data required by Gunrock. By providing Gunrock with the list of graph edges, it will initialize its graph datastructure on the GPU. After that, the primitive computation executes on the GPU and Gunrock outputs an array containing the lowest score to every image the provided source node is connected to. This may be an issue when dealing with very large amounts of images, but we do not address it in this thesis as it did not pose any problems with the image amounts we processed, and consider the limitation of Gunrock's output array for future work.

To finalize the query, we merely need to iterate this array to retrieve the n lowest scores, which correspond to the n most similar images to the source image (as we explained before, the lower the score, the more similar two images are).

Figure 4.3: *Query result using HOG features and the histogram comparison method, in a pool of 200 images. Left: Source image. Right: Most similar image to source image*

An example of a query result can be observed in Figure 4.3. In this example, we process 200 images using the HOG algorithm. Afterwards, we execute a query using the image on the left as the source image. When the computation is complete, we find that the most similar image to the source image is the one presented on the right, with a score of 67. This is merely one example to illustrate a query result. We present further examples of queries with features extracted from different algorithms (and their explanation) in Chapter 5, Section 5.7.

Note that, in order to compute a query, a user can use an SS Graph output by the solution right after its computation or, through our persistence system, he can load the SS Graph from a file and execute the query in the same manner as the one described above. In this case, before the query is executed, the image information (image IDs,

names, features and dates) is read from the files, the SS Graph is constructed once again in memory and the image features are loaded to the GPU. This functionality is important because it allows the execution of queries on images that were already processed, even if the program was stopped or the machines were to restart. It allows for the persistence of all the computations made, as well as further enrichment of the image database by executing and saving new computations.

Besides querying by providing an image that was already processed as the source image, it is possible to execute a query using any other image, even if it was not previously processed. The process works by providing the program with an image URL. The system will start by downloading the image, uploading it to the GPU and extracting features from it, much like the main workflow of our program. This process is similar to the one described in Section 4.4, the only difference being a different source node (that iterates and returns the already existing images, instead of processing a stream and downloading new ones), the exclusion of the GPU load node (as the features of the images that were already processed are previously loaded on the GPU) and the exclusion of the feature extraction nodes (as only the new image's features need to be computed, there is no need for the parallellization of the feature extraction step). In sum, the process can be described in the following steps:

1. If necessary, the SS Graph is loaded via file.
2. The query image is downloaded from the provided URL, uploaded to the GPU and its features are extracted. The image is indexed in the SS Graph as a node.
3. The modified source node iterates the SS Graph images, sending them to the feature matching node.
4. The feature matching node executes the comparison of each received image with the query image previously processed. After each computation, the corresponding similarity score is indexed in the SS Graph as an edge.
5. When all the comparisons have been made, the new SS Graph, which now includes the query image successfully indexed, can be used to execute a query normally, through the process described above.

4.6 Conclusion

In this Chapter we explained, in detail, the implementation of our system. We went over how and why it works. We also explained several implementation choices, such as the libraries, programming languages and frameworks that we opted to use. We presented the APIs and datastructures of our system, what purpose they serve and how they function, and finalized by presenting a detailed explanation of each step in our solution's pipeline, showing how our computations can be expressed as a flow graph. In the next Chapter,

we will evaluate, empirically and in detail, how the system we described here performs, both in terms of execution time and in terms of query precision.

Chapter 5

System Evaluation

This chapter intends to present an evaluation of the system we developed, measuring different metrics in order to identify performance and possible bottlenecks. A discussion of the obtained results is also presented. This evaluation allows us to identify the problems with our solution and design possible ways to improve it in the future. It also allows the reader to gain an idea of how our library performs in the task it was designed to do.

5.1 Metrics

In order to correctly and precisely measure our solution, we need to evaluate different metrics to allow us to identify how each component of our system behaves. Besides the measurement of different metrics, we need to vary our solution's configuration in order to evaluate which combination of algorithms and their parameters offer the best performance possible. Additionally, we need to experiment with different amounts of images to see how increasing the dataset size influences the program's behaviour, both in terms of execution time and in terms of memory usage. The metrics we measure in our evaluations are the following:

- Feature Extractor total execution time
- Time spent downloading images
- Time spent loading images into GPU memory
- Time spent extracting features for each algorithm used
- Time spent saving image data to files for each algorithm used, if applicable
- Time spent comparing image dates

- Feature Matcher total execution time
- Time spent per image comparison
- Time spent uploading graph (in Gunrock's format) to GPU
- Time spent executing Gunrock's graph primitive

The measurement of these metrics offer extreme detail to our evaluation and allows us to finely measure each component of our solution. It is important to note, however, that the total execution time of the whole pipeline is not equivalent to the sum of all of these metrics, as most of the components that execute the computations that these metrics measure are executed in parallel not only between each other but between themselves (*i.e.* one component can be executed concurrently by being operated by several threads, as we saw in the previous Chapters).

These measurements also allow us to investigate whether or not a component is a bottleneck and whether or not it suffers from starvation due to the slower execution of its predecessors.

5.2 Application

In order to test our system effectively, we need to develop an application that uses the algorithms we wish to test, with the ability to change different parameters and configurations. The application needs to extract features from the supplied images, compare images between each other and output a Similarity Search Graph for each algorithm used. The SS Graphs then need to be subjected to different queries. The application is extremely simple and merely executes the steps described above.

5.2.1 Application input: the YFCC100M Dataset

As seen previously, the input to our system is an image stream containing URLs. The source node reads the image URLs and is responsible for downloading the images to be later processed. Our implementation uses the Yahoo Flickr Creative Commons 100 Million (YFCC100M) dataset [35]. It is the largest public and free multimedia collection (includes images and videos), with around 99.2 million photos and 0.8 million videos from the image sharing platform Flickr [39]. It is used largely for computer vision research purposes, as the images are very diverse and contain several labels. It also includes *metadata* for most images, containing information about the images' identifier, the date they were taken and uploaded in, the geolocation of where the image was taken and a multitude more information. In our case, we only take advantage of the date taken field, however, it is possible to extract more information, should the application require it.

The dataset is provided in a 15GB compressed text file, and the source node processes it by converting the file into a stream and retrieving the necessary data. Each line in the text file contains all the information about a different video or image, with all its *metadata*.

5.2.2 Algorithms and Parameters

The application supports the SURF, HOG and ORB algorithms and can run them in any configuration desired. One can execute the solution using only one of the algorithms, or SURF/HOG, SURF/ORB, HOG/ORB or even all three simultaneously. We only require the specification of where the feature matcher for each algorithm resides (*hostname* and *port*).

Its also possible to specify whether or not the application should save image information and features to a file (as seen in Chapter 4). We also developed the application in a way that allows us to specify the amount of images we wish to process.

This way, we can easily change the application's configuration in order to meet the requirements of each test, without having to worry about the implementation of our system.

5.2.3 SURF Parameters

OpenCV's CUDA implementation of the SURF algorithm features several parameters that can be tweaked in order to offer more precision or speed, according to the requirements of the user. The parameters we tweaked are as follows:

Extended features If this parameter is set to false, SURF will compute 64-dimensional features, where as if its set to true, it will compute 128-dimensional features. Using the extended version of the descriptors offers more precision but slower extraction and matching speed. We chose to run SURF with the 64-dimensional features, as we found they offer reliable enough precision and significant speed-up over the 128-dimensional features.

Upright As we saw in Chapter 2, some algorithms are scale and rotation invariant. This means that, if an image is scaled or rotated, it will produce feature descriptors that describe the image independently of its scale or rotation. This helps with feature matching if the dataset contains rotated images, and allows the correct detection of objects or similarity calculation regardless of image orientation or scale, but it also introduces extra complexity which results in additional computations and, consequently, additional execution time. If, however, the dataset contains upright images, we do not require this additional computation. In an effort to help performance, and considering our dataset contains mostly upright images, we set this parameter to true, meaning the orientation of the images is not computed. This results in a much faster computation of feature descriptors.

Its important to note that OpenCV's CUDA implementation of SURF offers several more parameters, but they are more related to aspects such as the size of the features and the sensibility of the key points than they are related to performance. Considering this, we chose to leave these values at their default values.

5.2.4 HOG Parameters

Much like the case with SURF, OpenCV's CUDA implementation of the HOG algorithm also features several parameters. The parameters we tweaked are the following:

- Block size
- Cell size
- Block stride

These parameters correspond to the size, in pixels, in which the image is divided (images are divided into blocks, which are divided into cells). The smaller the values, the more detailed the feature descriptor will be (because the image will be divided in more parts), but it will also be larger. The default value for the block size is 16 pixels by 16 pixels, however, we found that this value resulted in features that were way too large for the scale of our system, resulting in memory issues. This lead us, after some experimentation, to increase the block size to 32 pixels by 32 pixels.

Because we increased the block size, and due to the fact that it must be proportional to the cell size, as they are dependant on each other, we also increased the cell size from *8x8* to *16x16*. Finally, to maintain proportionality, we also had to increase the block stride, to ensure the image features are correctly computed. Since we doubled the block size from *16x16* to *32x32* and the cell size from *8x8* to *16x16*, we also doubled the block stride from *8x8* to *16x16*.

We found that these changes, for our dataset, resulted in more manageable features while retaining much of the precision achieved when using the default values. Having smaller feature descriptors also results in faster feature matching, which is an extra advantage to our implementation.

5.2.5 ORB Parameters

In the case of the ORB algorithm we left all parameters untouched, since it was already developed to be as fast as possible. Considering we do not want to hurt its precision, nor speed up its execution, we do not require the tweaking of parameters in the case of this algorithm. All values were left in their default state, according to OpenCV's CUDA implementation of the algorithm.

Table 5.1: compute-0-{0,1} Hardware

Motherboard	ASUS P9X79 PRO
CPU	Intel Core i7-3930K Processor (6 cores, 12 threads, 12M cache, 3.2 GHz, up to 3.80 GHz)
RAM	64 GB DDR3 (1333Mhz)
GPU	NVIDIA GeForce GTX TITAN X 12 GB
Storage	WD Green WD30EZRX 3TB IntelliPower 64MB Cache SATA 6.0Gb/s 3.5"

5.3 Hardware and Configuration

Our system was designed to run on a cluster composed by four nodes available to us to aid in research and development. However, it is important to note that the system was designed to be scalable and easily adaptable to any type of configuration. With little effort, a user should be able to adapt the system to fit his hardware configuration.

In our case, however, the hardware we execute the solution in is composed by four rack-mounted computing nodes equipped with GPGPUs. It also has a master node used to issue jobs to the computing nodes. A diagram of the cluster can be seen in Figure 5.1.

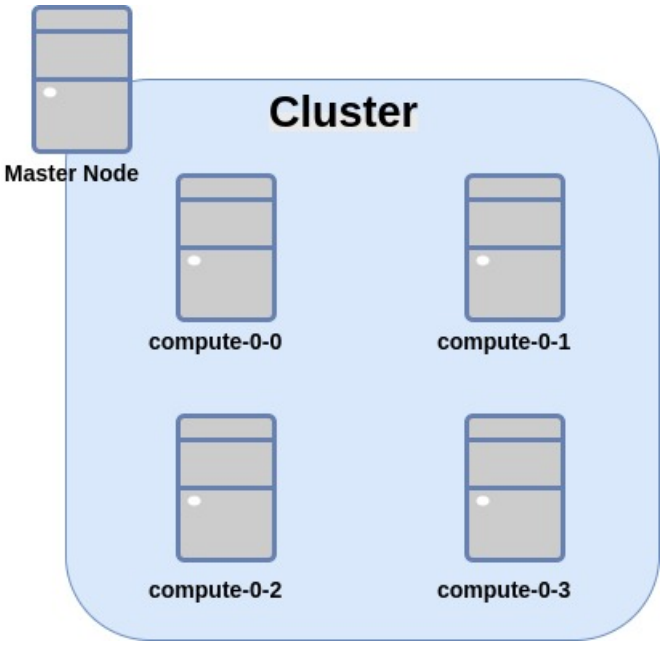

Figure 5.1: *Cluster Diagram*

All cluster computing nodes are very similar, however, *compute-0-2* and *compute-0-3* differ from *compute-0-0* and *compute-0-1* in that they are equipped with a different GPU. The hardware for these cluster nodes can be observed in Table 5.1 and Table 5.2. Also, all cluster nodes are interconnected using a Gigabit switch, with speeds up to 1000Mbps.

In order to map our solution to this cluster, we need to consider which cluster nodes need to run each role. Since we have one *feature extractor* and three *feature matchers*, the

Table 5.2: compute-0-{2,3} Hardware

Motherboard	ASUS P9X79 PRO
CPU	Intel Core i7-3930K Processor (6 cores, 12 threads, 12M cache, 3.2 GHz, up to 3.80 GHz)
RAM	64 GB DDR3 (1333Mhz)
GPU	NVIDIA GeForce GTX 1080 Ti 11 GB
Storage	WD Green WD30EZRX 3TB IntelliPower 64MB Cache SATA 6.0Gb/s 3.5"

mapping is simple. It is advantageous to have each machine execute the *feature matching* role in a dedicated fashion, since it is very computationally expensive. The one machine that is left is responsible for executing *feature extraction* for all three algorithms, which is more than enough. We found that having only one node extracting features is more than enough to keep up with the demand of the *feature matching* nodes, since they are significantly slower.

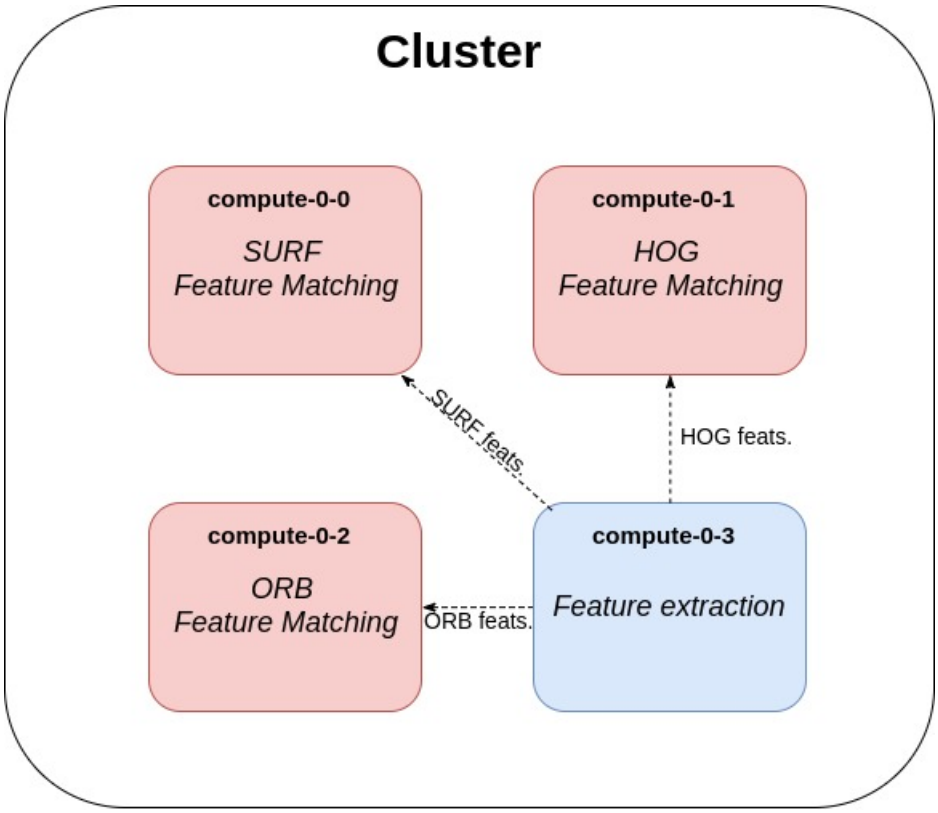

Figure 5.2: *Cluster Mapping*

A mapping of our solution to the cluster can be seen in Figure 5.2. As we can see, the master node is left out, due to the fact that it is not equipped with a GPU and that its introduction would introduce unnecessary complexity for little to no performance gain. Since most of our computations are executed on the GPU, we need only use machines equipped with GPUs. As these are the cases of the computing nodes, they are the only ones we use. As we can see, the first three computing nodes are constantly listening

for image features (each listening to the features regarding the algorithm its expecting) and, as they receive them, they process them by performing feature matching in order to calculate the similarity scores and use the scores to construct the SS graph. The last computing node processes the images and outputs feature descriptors, sending them through the network to the other nodes.

5.4 Single Algorithm Tests and Results

In this section we test each algorithm individually, using two separate machines. One machine performing the *feature extractor* role and another performing the *feature matcher* role.

We execute tests for 500, 1000, 5000 and 10000 images. Although these numbers seem small, they entail in 124750, 499500, 12497500 and 49995000 image comparisons, respectively. Additionally, processing more images than that starts to produce memory issues, an issue we do not tackle in this thesis (and consider for future work, as explained in Chapter 6). We do, however, tackle the problem of comparing as many images as possible. A system that is able to compare many thousands of images per second means a fast system when performing a similarity query, which is the focus of this thesis.

In the first round of tests, we had every persistence capability enabled (*i.e.* saving image information, features and scores), whereas in the second round, we disabled these functionalities in order to better observe if they have an impact on overall performance.

5.4.1 Feature Extraction with persistence enabled

Figure 5.3 shows the total execution time of the *feature extractor* role in seconds, in function of the number of images processed. We can see an exponential increase in execution time, for every algorithm, as we process more images. When processing small amounts of images (500 and 1000), the execution times of all algorithms are very similar, due to the significantly smaller amounts of comparisons that need to be made. When moving on to larger amounts of images, however, we see a significant difference between the execution time of SURF and ORB when compared to the execution time of HOG, which is expected as HOG is the overall slower algorithm of all three. A question arises, however. Why do we see an exponential increase in time as the number of images get bigger, instead of a linear increase? An exponential increase would be expected in the *feature matcher* due to the number of image comparisons it has to make (which increases exponentially with the amount of images processed), but not so much in the *feature extractor*, which only has to extract features from images. Additionally, it is expected that HOG is slower at feature matching, but not at feature extraction. This happens due to several reasons (such as the saving of image features to files slowing down the overall process), which we will explore in further detail below. For now, it is important to note that while the total execution time of the *feature extractor* is lower than the total execution time of the *feature matcher*, no

problems arise whatsoever, as long as the *feature matcher* does not starve and the *feature extractor* does not delay sending it the image features it computes.xss

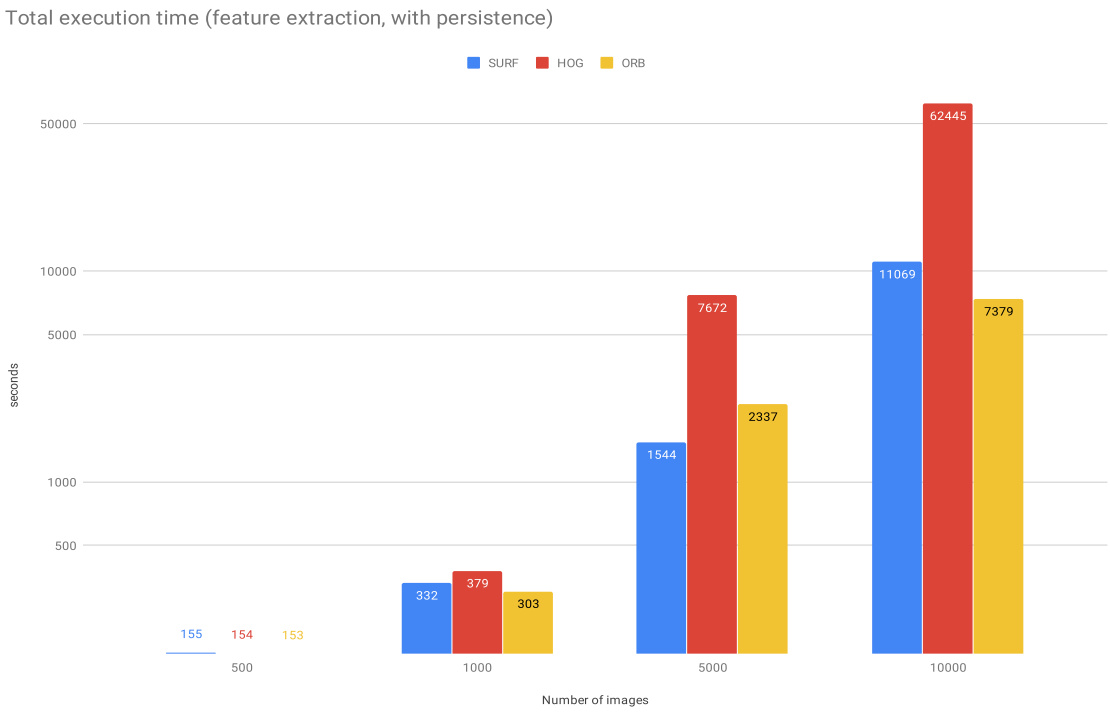

Figure 5.3: *Total feature extractor time (persistence enabled), in function of the number of images processed (logarithmic scale).*

Figure 5.4 shows the average time spent in the feature extractor, per image, for the different amounts of images we tested against. It is clear, at first glance and for all algorithms, that the more images the *feature extractor* has to process, the slower it becomes at processing them. This is due to a few factors. First of which, when there are more images being processed, the *feature matcher* eventually becomes slower and, considering its network source node has a limited buffering capacity, it eventually stops receiving images until it has processed the ones it currently holds and has space for more. This means that the *feature extractor* must wait before sending the features it just processed until the *feature matcher* is ready for more. It is important to note that the time measured in Figure 5.4 is the total execution time of the *feature extractor* divided by the amount of total images processed. Since the total execution time only stops measuring once the *feature extractor* finishes sending all of the extracted features, we obtain larger values for larger image numbers, because in those cases, the *feature matcher* will stall receiving images from the *feature extractor*.

This is further evidenced by the graph in Figure 5.5, which shows the average time it takes for an image to be downloaded, pre-processed, loaded to GPU and have its features computed. As we can see, disregarding the time the *feature extractor* is waiting for the

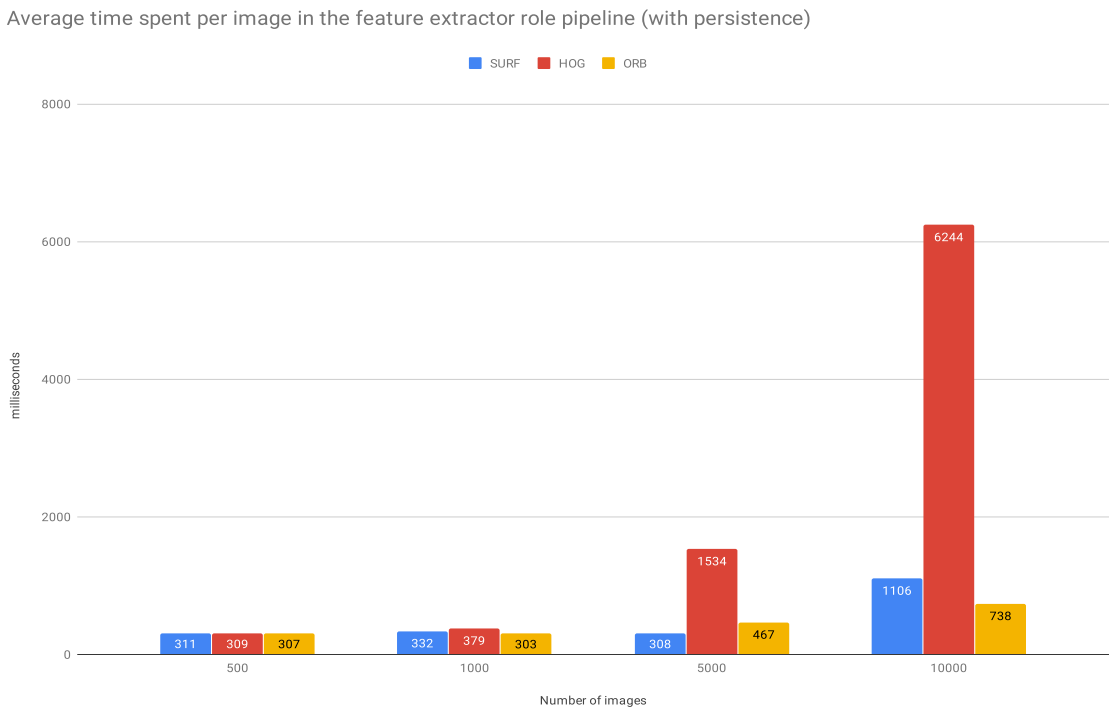

Figure 5.4: *Average time spent in the feature extractor (persistence enabled), per image*

feature matcher (as well as other nodes such as time load, time comparison and feature file save), the time it takes for an image to arrive from the stream and have its features computed is more or less equivalent independent of algorithm or number of images (with the exception of the HOG algorithm, which is significantly faster at extracting features when dealing with larger image numbers). The disparity between Figure 5.4 and Figure 5.5, then, suggests that it is not really the feature extraction step that is slowing down the overall *feature matcher* role. Other steps this role executes, such as feature file writing, significantly slow down the overall pipeline, as we will see subsequently.

There is a significant increase in processing time when processing 10000 images. This is due to the second factor that influences the *feature extractor* time when dealing with larger amounts of images: CPU sharing. Because there are several threads working concurrently to execute each step of the *feature extractor* pipeline (*i.e.* image download, image GPU load, image feature extractor, image date extraction, image date comparison, image feature file save), the more images they have to process, the more CPU time they will have to share, which slows each step down. It also hinders the feature extraction step because, even though its executed on the GPU, it still has some computations on the CPU (such as TBB-related computations and some other calculations it needs to perform). Overall performance is especially penalized (when dealing with larger amounts of images) due to the image feature file save node, because it becomes increasingly slower at writing to the file when dealing with more images and, eventually, starts slowing down other

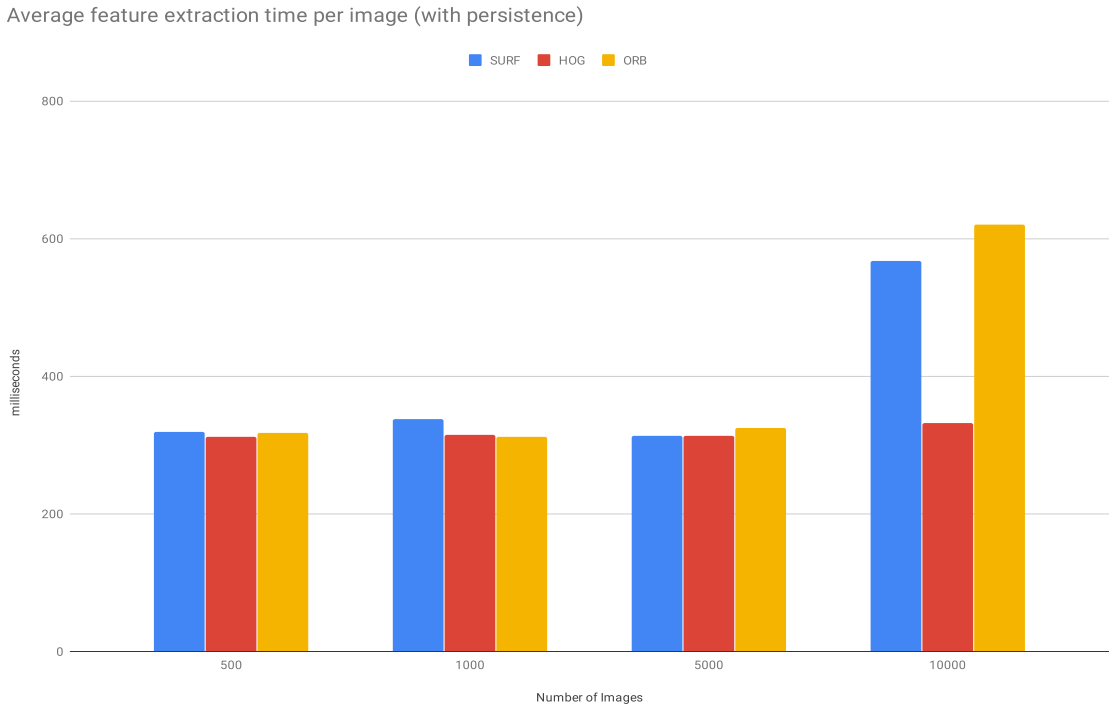

Figure 5.5: *Average feature extraction time (persistence enabled), per image*

pipeline nodes. This is a snowball effect, which slows down the whole pipeline.

All of this is complemented by the evidence in future subsections, where we execute tests without the persistence functionalities enabled.

5.4.1.1 Pipeline steps comparison

Table 5.3 shows a comparison between the different steps in the feature extraction pipeline, for different amounts of images. The times shown are in milliseconds and represent the average time taken to complete each step.

We can see that all nodes have similar times for different numbers of images processed and for different algorithms, except for the feature extraction and image feature save nodes, which makes sense because these are the only nodes that are influenced by the type of algorithm that is running. Another exception for the similarity of times are the times of the nodes when processing 10000 images, where most times increase when compared to lower image numbers. This is due to the CPU sharing phenomenon we explained earlier and we should encounter a more balanced version of this table if we turn off the persistence features of our system, as evidenced by the amount of time taken to save image features. When dealing with 10000 images, we can see an increase in time for all nodes when compared to when dealing with 5000 or less images, except the GPU load node. This can be attributed to the fact the the GPU load node uses almost exclusively the GPU and, thus, does not suffer from this CPU sharing issue. As explained

Table 5.3: Feature Extraction pipeline (with persistence) steps average execution time, in milliseconds.

	Algorithm	500 Images	1000 Images	5000 Images	10000 Images
Source Node	**SURF**	311	331	307	551
	HOG	308	311	315	327
	ORB	307	302	309	551
GPU Load Node	**SURF**	5	4	4	3
	HOG	4	4	1	4
	ORB	4	4	4	3
Image Date Load Node	**SURF**	0.035	0.035	0.022	0.033
	HOG	0.038	0.039	0.008	0.043
	ORB	0.034	0.034	0.022	0.039
Feature Extraction Node	**SURF**	3	3	2	13
	HOG	0.6	0.6	0.2	1.4
	ORB	7	6	12	66
Image Date Comparison Node	**SURF**	0.008	0.007	0.005	0.025
	HOG	0.009	0.007	0.004	0.005
	ORB	0.009	0.007	0.005	0.022
Image Feature Save Node	**SURF**	24	24	24	148
	HOG	65	65	63	65
	ORB	3	3	3	35

before, although the feature extraction node is executed on the GPU as well, it does have some computations it needs to do on the CPU, hence why it is also slowed down and does not remain unaffected like the GPU Load Node. Also, it is important to note that HOG is not affected as much by this CPU sharing effect. This is due to its much faster feature extraction, meaning it has more free CPU time than is the case with other algorithms, which results in a better sharing of processing time between all nodes, thus mitigating this problem.

This table, along with the times for feature matching shown below, also shows the significant difference between comparing image dates and comparing image features (and, for that matter, between extracting image dates and extracting image features). This is the reason we execute date comparison in the *feature extractor* rather than in the *feature matcher*, because this step takes dozens of microseconds while feature matching takes tens of milliseconds and does not influence the overall performance of the *feature extractor*.

We can also observe that the time it takes to save image features is significantly higher than any other time in the pipeline (with the exception of the source node). This shows that we need to consider carefully whether or not we want to save image features, and evaluate whether or not it slows down the overall solution, as we claimed earlier. This will be evaluated in Subsection 5.4.3 and Subsection 5.4.4, where we turn off the feature save ability of our program and evaluate empirically whether or not it has an impact on overall performance.

5.4.2 Feature Matching with persistence enabled

Figure 5.6 shows the total execution time of each algorithm for the *feature matcher* role, in function of the number of images processed. Note that the total execution time is always higher than in the case of the *feature extractor*, for the same algorithm and number of algorithms processed. This is expected, as we know that the *feature matcher* can not finish without the *feature extractor* completing its work first. Additionally, we see an exponential increase in the total execution time, for every algorithm, as we increase the number of images. This is another expected fact, because as we increase the number of images, we exponentially increase the number of comparisons that need to be made. For instance, when processing 500 images, we will perform 124750 comparisons and, when processing 10000 images, we will perform 49995000 comparisons. We went from 500 to 10000 images (a 20x increase) and from 124750 to 49995000 comparisons (a 400x time increase). Even if the average time it takes to complete a comparison remains the same (which usually increases), the total time will increase exponentially as well. Additionally, we can see that the feature matching time is more or less similar between the SURF and ORB algorithms (because these algorithms have very similar matching methods), but always higher in the HOG algorithm, due to its significantly slower matching process. Finally, when dealing with small amounts of images, the total *feature matcher* time is very closely tied between all algorithms, because the small amount of comparisons that are made do not allow for the feature matching method to have enough influence on the overall execution time.

Figure 5.7 shows the *feature matcher's* average comparisons per second, in function of the number of images.

When the amount of images is lower, the number of comparisons per second is also lower, because of the image download step. With a low amount of images, the feature matching step executes at a rate that is much faster than the rate at which images are downloaded, which results in some starvation initially. It is only until the feature matching step becomes slower from the higher amount of images being processed that the download step can keep up with it and supply images in time to avoid starvation. This is why, when dealing with 1000 images, the number of image comparisons per second is almost twice that of the number of image comparisons when processing 500 images. We can also see significant more comparisons per second when processing 5000 images than when processing 1000 images (almost 3 times higher for SURF and ORB and around 1.3 times higher for HOG). This is when the program reaches its "sweet spot"where the download step is sufficiently slower than the feature matching step to avoid starvation and the feature matching step has a manageable amount of images so that it does not become too slow, which is what happens when processing 10000 images. Here, the number of image comparisons per second stabilizes back to what it was when dealing with 1000 images. Now it is not due to the download step, which still provides sufficient images

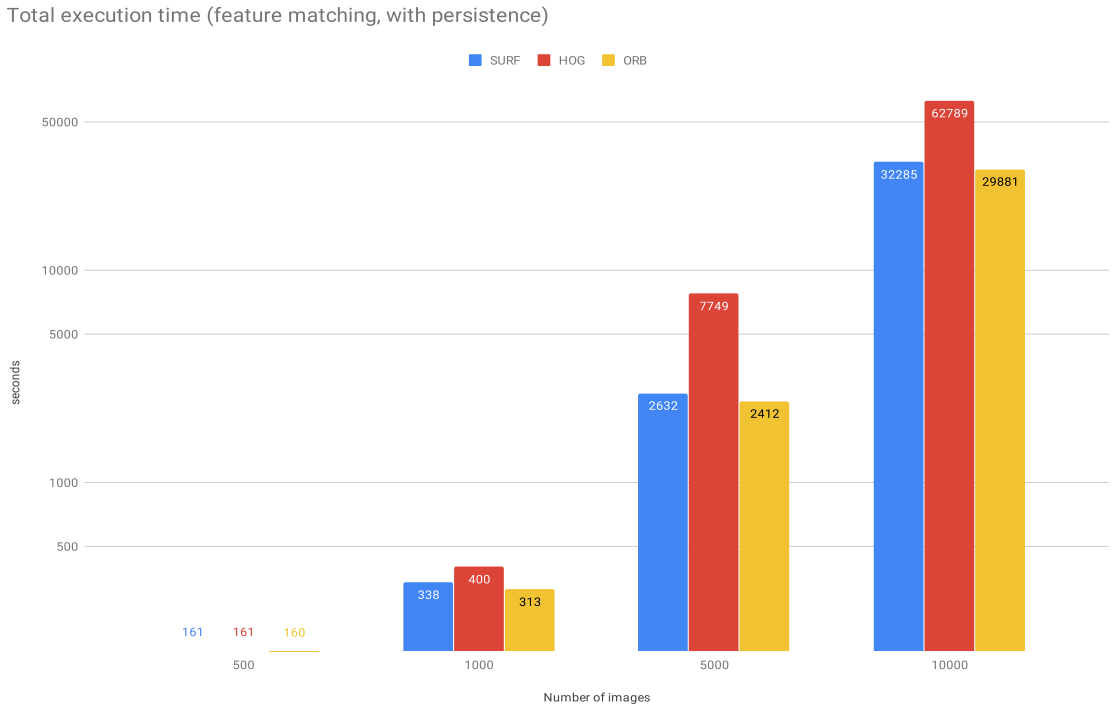

Figure 5.6: *Total feature matcher time (with persistence), in function of the number of images processed (logarithmic scale).*

for the feature matching step to process, but due to the fact that the increased amount of images slows down feature matching, as several more comparisons need to be made. In any case, when processing 10000 images, we see comparison rates of almost 2000 images per second, which is great performance if we consider the image comparison focus of the system. If we were to perform a similarity query with a new image, against a library of 10000 images, we know it would only take around 5 seconds for all the necessary comparisons to be made, which is a more than acceptable trade, considering the significant advantages it brings.

5.4.3 Feature Extraction with persistence disabled

In order to properly corroborate the arguments made above regarding the influence of the feature save node being turned on (*i.e.* saving image features and other information to file), we need to repeat the tests with this option disabled and evaluate whether or not they have an actual impact on performance.

The first graph, presented in Figure 5.8, shows us, as before, the total execution time for the *feature extractor* role pipeline, in seconds. As before, we still see an exponential growth in the time it takes to process images in the *feature extractor* as we increase the number of images processed. Like before, this exponential increase is due to the fact that the *feature extractor* sometimes has to wait for the *feature matcher* to accept more images,

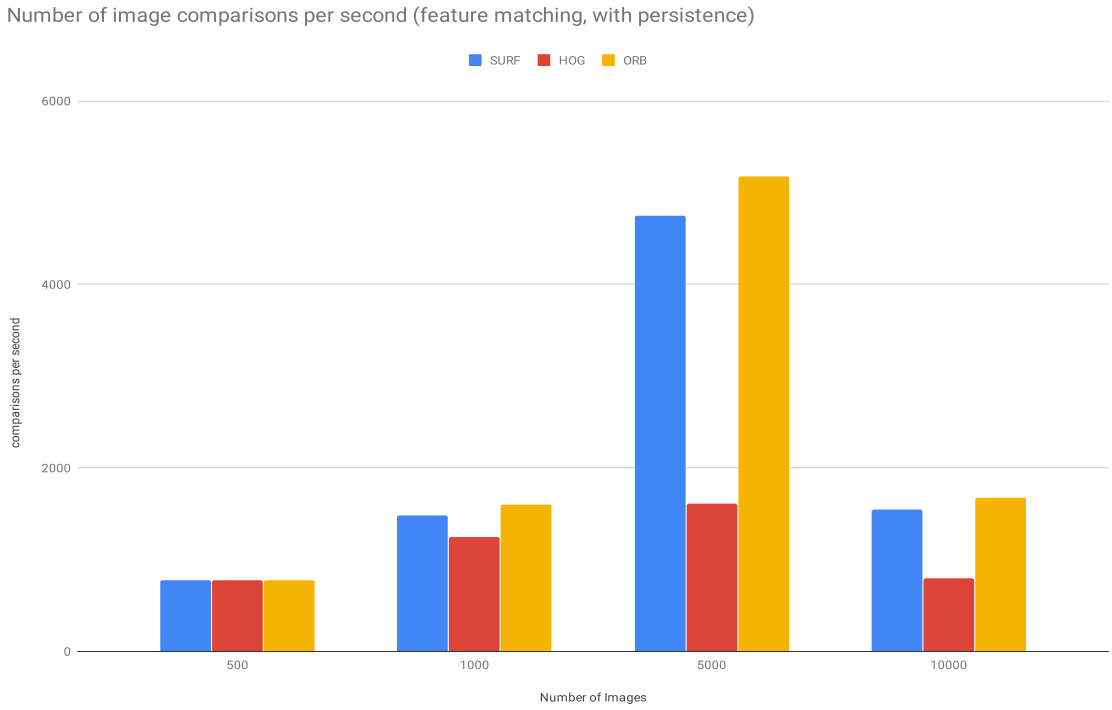

Figure 5.7: *Feature matcher average comparisons per second (persistence enabled)*

as it becomes busier. This is a process that happens more often when larger amounts of images are being processed and, thus, the feature matching process takes longer.

We do, however, see a big difference between this graph and the graph present in Figure 5.3. The difference is the total execution time between each algorithm for the same number of images in the case where persistence is enabled and the case where persistence is disabled. In the case of SURF, for 10000 images, the total execution time is almost 3 times less in the case where persistence is disabled versus the case where persistence is enabled. In the case of ORB, the difference is significantly different (around 440 milliseconds), which is quite expected. ORB features are significantly less memory intensive, which means writing them to a file is faster than is the case for SURF or HOG features, and so the feature save node does not introduce that many delays in the pipeline. For HOG, the difference is not so significant either. This is due to the fact that HOG has a very slow feature matching process, when compared to other algorithms. This means that the *feature extractor* waits significantly more time for the *feature matcher* to be ready to receive more features than in other algorithms and has plenty more CPU time to spare. This spare CPU time can be used for the writing of image features to a file, without too big an impact on performance.

When analysing the times for fewer images, the difference between total times between the case with persistence enabled and persistence disabled is not so significant, because the lower amount of images does not allow the feature save node to introduce

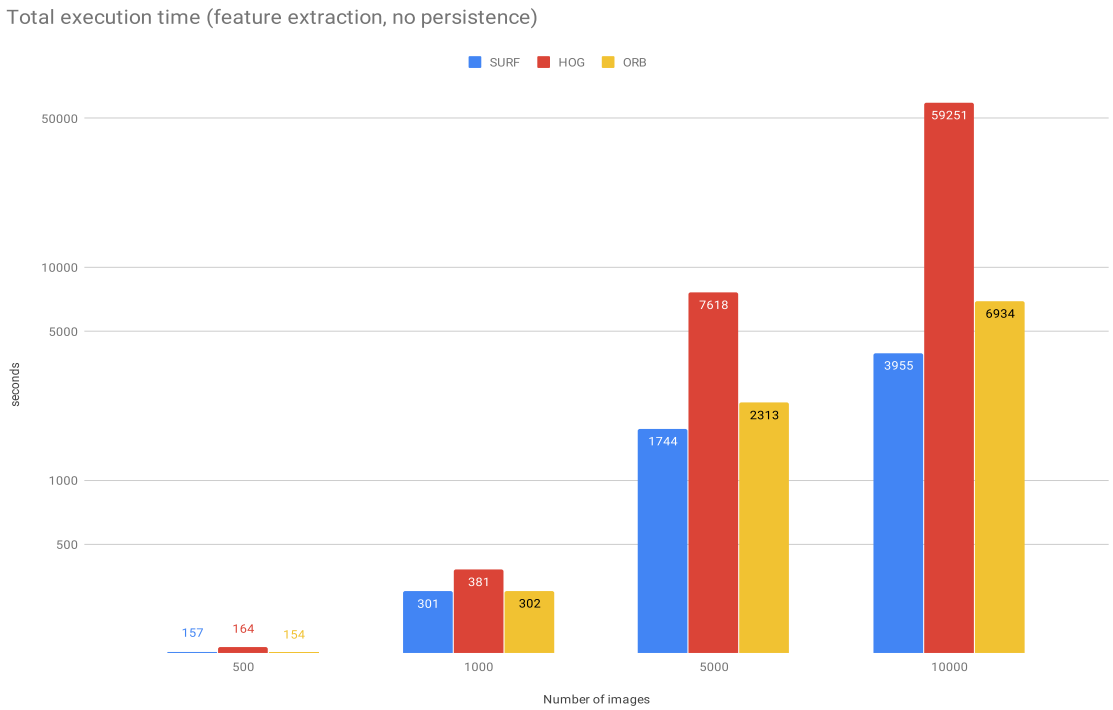

Figure 5.8: *Total feature extractor time (persistence disabled), in function of the number of images processed.*

many delays in the system. This can be further evidenced in Figure 5.9, that shows the average time each image spends in the whole *feature extractor* role pipeline. Between this Figure and Figure 5.4, the only significant time difference is when processing 10000 images, which makes sense, considering the eventually slower feature matching process, which in turn slows down the feature extractor and, if the feature extractor is also busy with saving image features, the whole feature extraction process will be slowed down. But in order to properly evaluate whether or not a slower feature extraction process (*e.g.* with persistence enabled) also results in a slower feature matching process, we need to take a look at the feature matching results with persistence disabled and compare them with the feature matching results with persistence enabled. This is what we cover in the next Subsection.

This test gives us a more precise view over the actual time it takes to process an image, without the delay introduced by saving features, but it still does not allow us to compare the performance of the algorithms at the task of extracting features. Remember, this graph shows us the total time for the execution of the *feature extractor* role pipeline, which includes more than extracting features and is also delayed by the *feature matcher* role. For that, we need to analyse the time of the feature extraction node of the pipeline exclusively, as we did in Figure 5.5 (in that graph, we also consider the download and GPU load steps time, as they are required before extracting features). Repeating this step for the

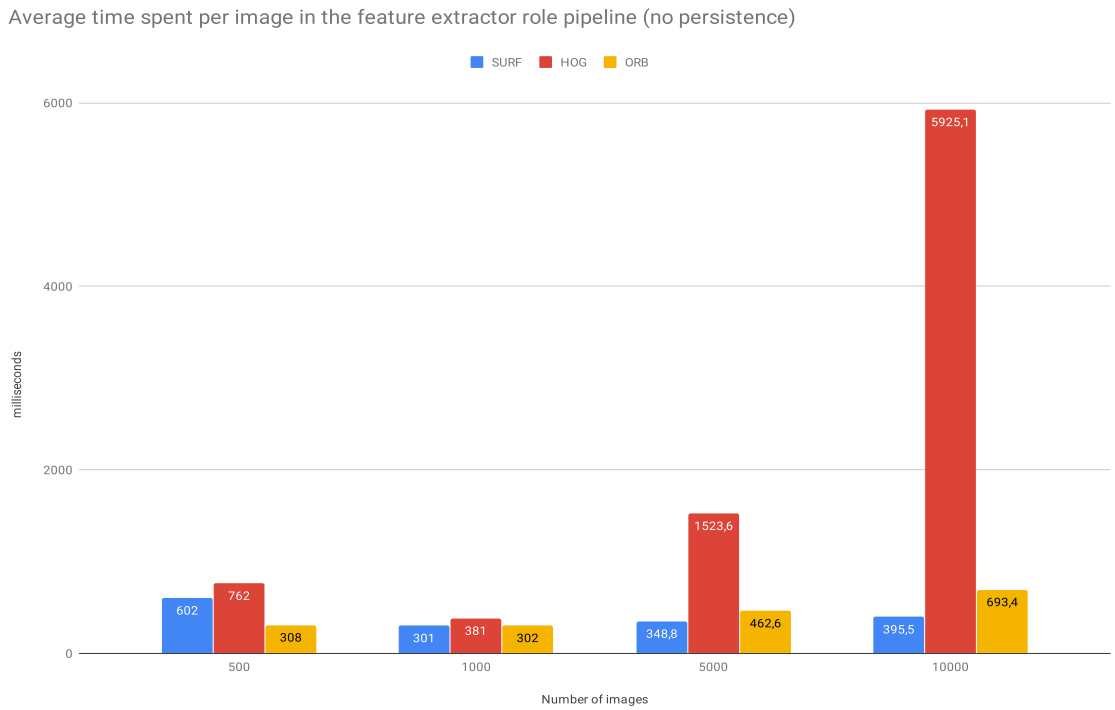

Figure 5.9: *Average time spent in the feature extractor (persistence disabled), per image*

tests with no persistence enabled, we constructed a graph that shows the average feature extraction time, per image, for the different numbers of images processed and algorithms used. The times shown, in milliseconds, represent the average time it takes for an image URL to be retrieved from the stream, the respective image downloaded, uploaded to the GPU and have its features computed. The resulting graph is shown in Figure 5.10. Here, the average feature extraction time is very similar and quite independent of the number of images processed. This is due to the no longer present CPU sharing phenomenon introduced by the saving of image features and information. This means that when processing larger amounts of images, there is no longer a step (feature save node) slowing down the other steps, which can execute normally and without delays. This is the case of feature matching with persistence enabled (Figure 5.5), where there is a clear delay when processing 10000 images, when compared to the processing of fewer images. With persistence disabled, however, this is not the case. Hence, we can safely conclude that the delay presented by the saving of image features is in fact significant, and becomes worse with larger amounts of images.

5.4.3.1 Pipeline steps comparison

Table 5.4 shows us a similar table to what we have seen before. It contains the different times, in milliseconds, for each node of the feature extraction pipeline, and for each algorithm. As we can see, there is no significant time increase in the nodes from 5000

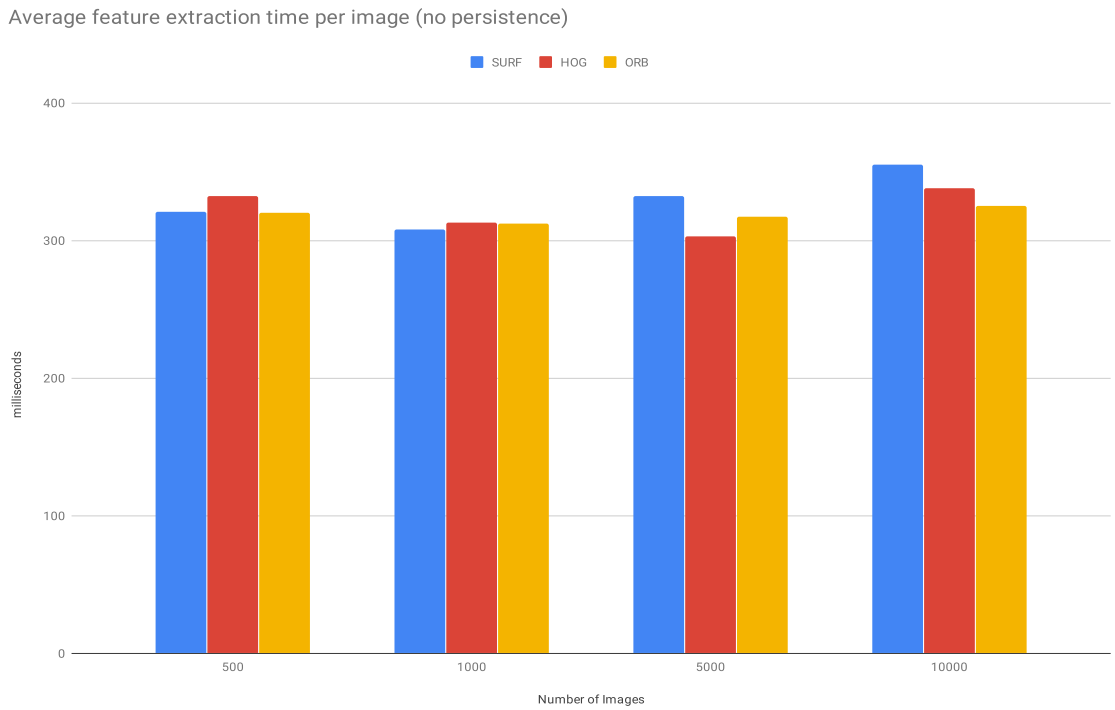

Figure 5.10: *Average feature extraction time (persistence disabled), per image*

to 10000 images, for any algorithm, as there was before. This shows that turning off the feature save and image information save nodes significantly improves performance. Now, CPU sharing is no longer a significant issue and each node can execute its computation in a more regular time, consistent to what we observe when dealing with fewer images.

This table shows the peak performance of our system's feature extraction role. It means we can extract features at an extreme high rate. Without considering the source node (we also offer the optimization of processing images that are already downloaded, although its not our focus), we can extract around 125 SURF and HOG features per second and around 77 ORB features every second. This is extremely good performance, and follows from our design goal of processing images with high performance. A user can still benefit from not saving image features, if all he needs is the actual graph. If a user writes the graph (merely the image identifiers and the scores between them) along with the file that maps image names to image identifiers, he has all the information he needs to extract image similarity information regarding the images. He can also read from that graph and add more data to it, without having to deal with the features of all the images that were already processed. If, however, a user needs to compare newer images with past images, it may be wiser, in the long term, to save the image features for later comparison. It may be better to take the performance penalty right away, instead of running the whole process again, which will mean, undoubtedly, significantly more overall runtime.

Table 5.4: Feature Extraction pipeline (no persistence) steps average execution time, in milliseconds.

	Algorithm	500 Images	1000 Images	5000 Images	10000 Images
Source Node	**SURF**	314	301	326	347
	HOG	328	309	299	330
	ORB	309	302	307	312
GPU Load Node	**SURF**	4	4	4	4
	HOG	4	4	4	5
	ORB	4	4	4	4
Image Date Load Node	**SURF**	0.034	0.034	0.038	0.045
	HOG	0.036	0.036	0.038	0.048
	ORB	0.035	0.034	0.039	0.045
Feature Extraction Node	**SURF**	3	3	2	4
	HOG	0.6	0.5	0.74	3.1
	ORB	7	6	6	9
Image Date Comparison Node	**SURF**	0.008	0.007	0.005	0.005
	HOG	0.009	0.009	0.005	0.005
	ORB	0.008	0.007	0.005	0.004

5.4.4 Feature Matching with persistence disabled

Figure 5.11 shows the total time for all algorithms, in seconds, of the *feature matcher*, in function of the number of images processed. The graph is in logarithmic scale. This graph is very similar to what we observed in Figure 5.6, except in this case persistence is disabled. If persistence is disabled, there is a performance gain for all algorithms when processing higher amounts of images, although it is not as significant as is the case in the *feature extractor*. This also shows that, although the *feature extractor* influences the performance of the *feature matcher*, its significantly slower execution does not mean a significantly slower execution of the *feature matcher*.

This time difference between the case with persistence enabled and the case with persistence disabled, however, becomes larger and larger with higher amounts of images, which still means a clear impact on the performance of the whole pipeline, due to the persistence functionalities of our system.

Figure 5.12 shows the average number of image comparisons per second for every algorithm, in function of the number of images. This graph is very similar to what can be seen in Figure 5.7. Here, however, there is a big difference. When processing 10000 images, the number of SURF and ORB comparisons per second almost double in comparison to what it was before (with persistence enabled), and closely match the performance obtained when processing 5000 images. This is very significant, because jumping from 2000 image comparisons per second to 4000 image per second is a clear performance gain. There is no significant performance gain in HOG's image comparisons per second, although there is still a small gain. HOG has a very slow feature matching

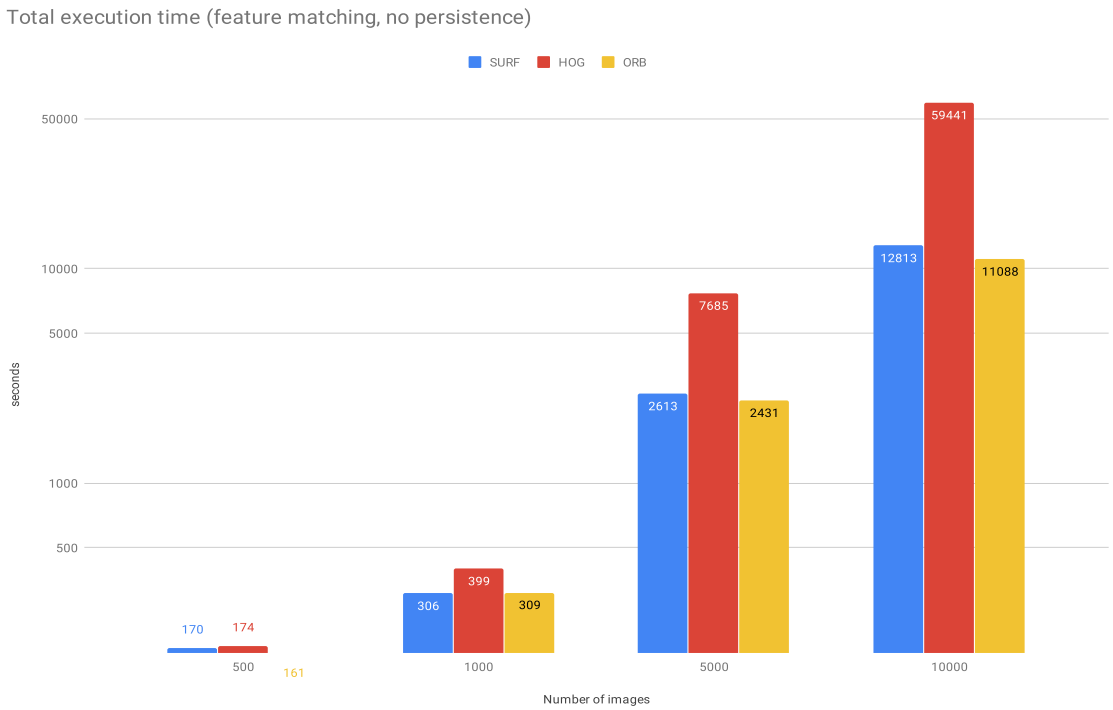

Figure 5.11: *Total feature matcher time (no persistence), in function of the number of images processed (logarithmic scale).*

process nonetheless, and benefits little from choosing not to save image features.

This is another graph that sells the notion that choosing to save image features has a very significant impact on the overall performance of our system and is a functionality that should be enabled carefully, as previously mentioned.

Now, however, we can safely show the full performance power of our library. When processing large amounts of images, we still obtain around 4000 image comparisons every second for SURF and ORB, and around 850 comparisons per second for HOG. The ability to compare 4000 images every second is a very powerful one, and is aligned with the high performance image comparison, similarity search design goal of our system.

5.4.5 Summary

The main and most obvious conclusion we can withdraw from these tests is the fact that choosing to save image features to a file, for later processing, has a significant impact on the overall performance of our system. It is clear from this fact, that a user needs to reason carefully about such a choice, which should only be made if he knows for certain that there will be a need to perform comparisons between new images and old images that were previously processed. If, however, a user only needs to process a certain amount of images and knows he will not need to use them again for other comparisons in the future, he should opt for not saving image features. In any case, there is always the possibility

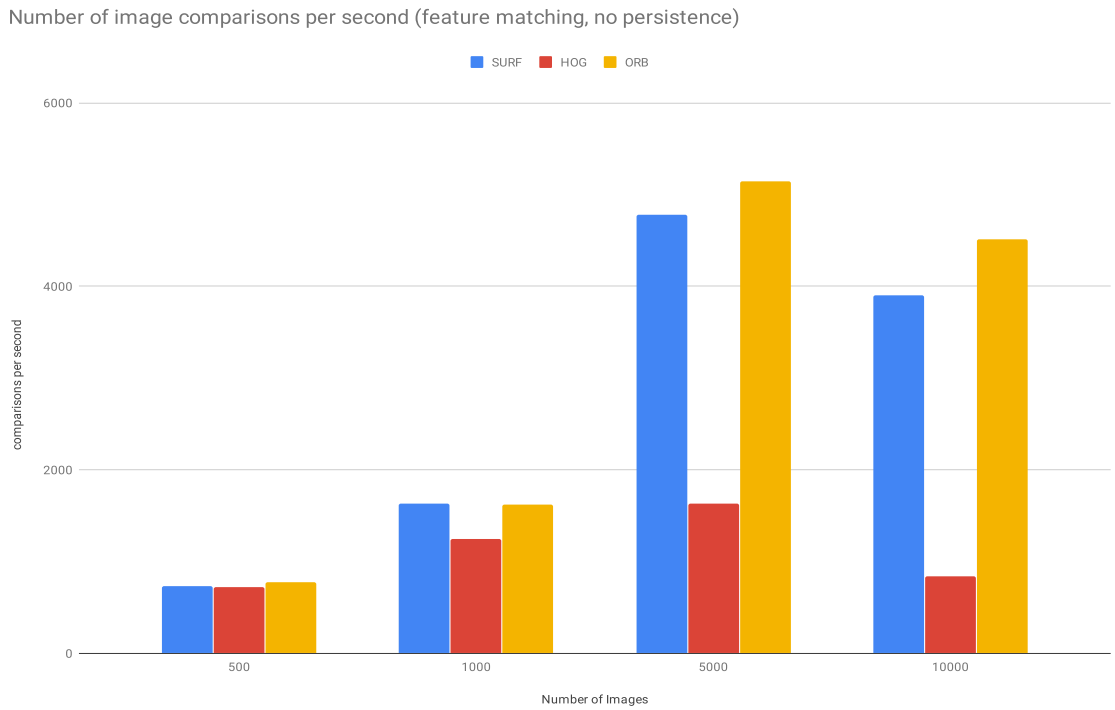

Figure 5.12: *Feature matcher average comparisons per second (persistence disabled)*

of saving the graph information to a file (image identifiers, names and scores), which is a process that does not take much time at all. This way, one can always consult the scores obtained at a later point in time, or even perform further queries on the graph.

An additional fact that can be extracted from the tests, is the much better performance of SURF and ORB, in the overall pipeline. Although HOG has a quicker feature extraction process, it has a much slower matching process which, in turn, slows down the whole solution. Even considering that HOG typically has more precise features, there needs to be a choice whether or not to enable it, taking into consideration the precision of the matches the user needs to obtain. SURF still offers quite precise comparisons, and while ORB is the least precise of the three, it still has some useful properties and uses, such as low memory impact.

There is another conclusion we can take, which is the fact that the source node is also an expensive step. Downloading every image takes time and, while the number of images that have been processed is still low, it influences the performance of the library. It is only when the number of images increases above 1000 that the feature matching and extraction pipelines become slow enough to mask the time the source node takes. There is, of course, an optimization that can improve this. Instead of processing a stream of images, one can modify the source node to read from images that were previously downloaded, which significantly speeds up the whole process. Although the focus of this thesis is to process an image stream and extract similarity information regarding the images that

the stream provides, this is a useful functionality that we included in the system. In can also be used as a sort of cache where, if the system verifies that a given image from the stream is already present in the machine, it will simply read from the machine instead of downloading and writing the file again, redundantly. This can also be useful if a user that chose not to write image features to a file, needs to process a subset of the images that were previously processed in order to compare them against newer images and update the graph. Although the features will need to be extracted again, atleast the most time consuming step of the feature extraction pipeline will be significantly sped up.

5.5 All Algorithms Tests and Results

This section tests the whole solution, taking advantage of its full capacity. Here, we use the mapping shown in Subsection 5.3. Using four cluster machines, one as the *feature extractor* (extracting features from all three algorithms) and three *feature matchers* (each matching features for each algorithm).

In this test, we do not present results for the feature matchers. This is due to the fact that their results are virtually similar to the ones present in the single algorithm test in Section 5.4. The reason for this is simple, and resides in the fact that both in these tests and the single algorithm tests, each feature matcher runs on a single machine and matches features for a single algorithm. There could be a difference due to different, worse performance in the feature extractor, resulting in starvation of the feature matching nodes, but after assessing that this does not occur, we chose to omit the feature matching results for the sake of brevity.

There is, however, a difference in performance for the feature extraction node, which makes sense considering this node is now doing three times the work it was doing in the single algorithm test. Now, the *feature extractor* has to compute features for all three algorithms, and send them along to the respective *feature matcher*.

The tests were run without persistence enabled, due to the fact that we are already aware of its impact on the performance of the feature extractor. Here, its impact is no different and quite similar to the one presented in the single algorithm tests. The only difference is that, here, the *feature extractor* extracts features from two more algorithms than before, but since the execution time of each feature extraction step is quite negligible when compared to the source or the feature save node, the impact on performance derived from the persistence functionality here is very similar to what it was in the single algorithm tests. This is the reason why, once again, we choose to omit these values.

Figure 5.13 shows the total *feature extractor* execution time (with all algorithms enabled), in function of the number of images. Its important to note that the way the time is measured here is by starting the timer when the program first executes and finishing it once all its tasks are complete. This means that the times for the *feature extractor* with all algorithms enabled will always be as slow as the slowest algorithm. From before, we know that the overall slowest *feature extractor* is for the HOG algorithm. This is why

these times are similar to the ones obtained in Figure 5.8, albeit slightly higher. The extra time is attributed to the extra work the *feature extractor* has to execute, because it is now extracting features using three algorithms instead of just one. There is an advantage to doing processing this way instead of a single algorithm at a time, which is the fact that images only have to be downloaded and uploaded to the GPU once (instead of once for every feature extractor/algorithm combination). Additionally, by extracting features one at a time (using the single algorithm approach) we will eventually perform several redundant computations, such as the extraction and comparison of image dates, which obviously do not depend in any way on the algorithm used.

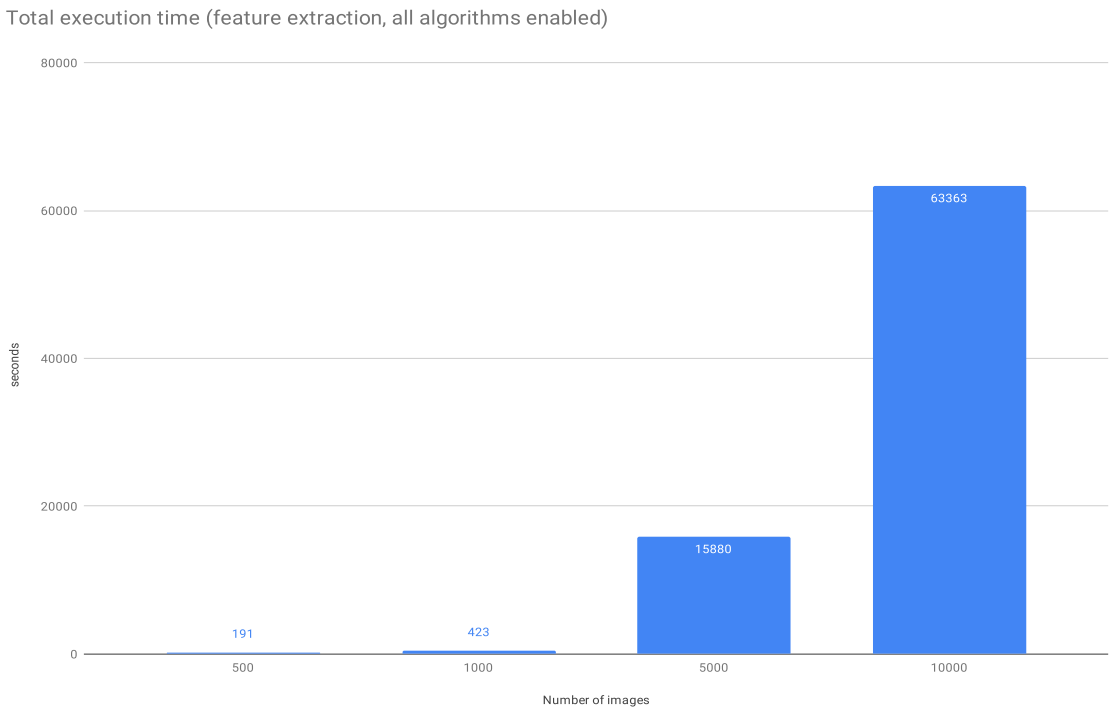

Figure 5.13: *Total feature extractor time (all algorithms enabled), in function of the number of images processed.*

Using the all-algorithm feature extraction approach, we are able to cut down on these extra computations and extract features from all the algorithms we desire right away. Additionally, regarding feature matching, there is no extra work involved because, as with the single-algorithm approach, there is still one *feature matcher* per algorithm, each residing on a different machine. This means their computations are not influenced by each other and execute independently of the amount of algorithms being used simultaneously.

The next graph, shown in Figure 5.14, contains the average time each image spends in the whole feature extraction pipeline, in milliseconds. As before, this time is as slow as the slowest algorithm, which means these times are similar to HOG times in Figure 5.9. The times here are slightly higher, as before, due to the extra computations that are being made. But, similarly to the conclusions we withdrew before, the time increases

exponentially with the amount of images processed due to the two factors we previously mentioned: waiting for the *feature matcher* to be ready for more image features and the CPU sharing effect.

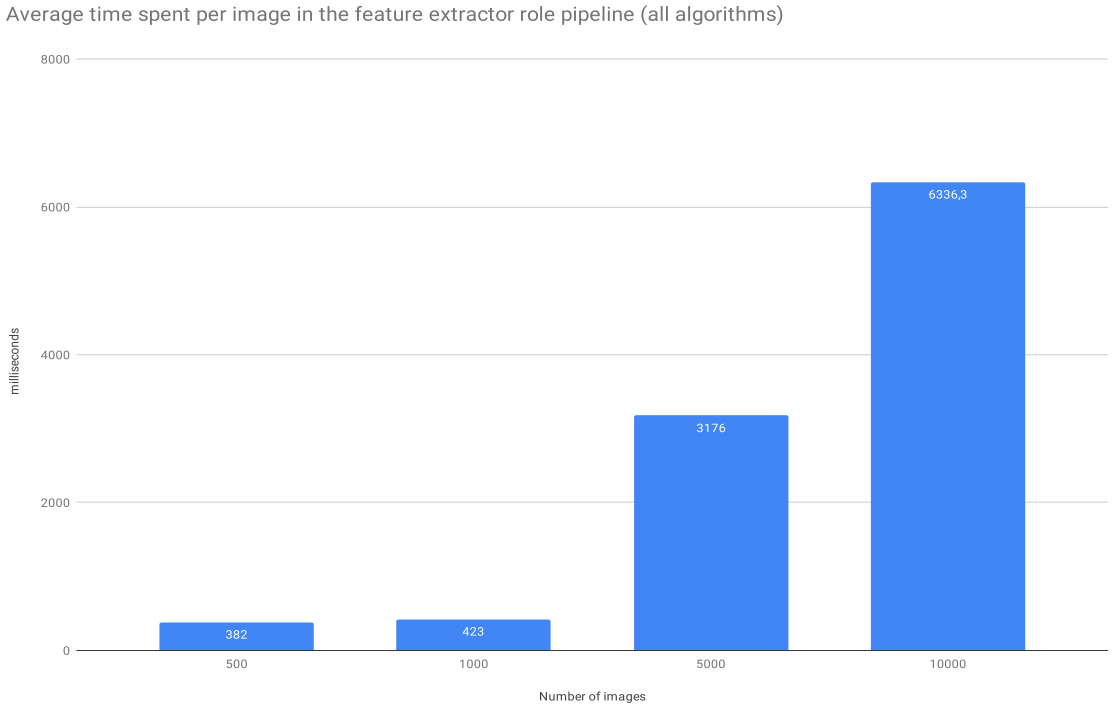

Figure 5.14: *Average time spent in the feature extractor (all algorithms enabled), per image*

Finally, Figure 5.15 presents the average feature extraction time, for each algorithm, in function of the number of images. The times are, as before, presented in milliseconds. Similarly to Figure 5.5 and Figure 5.10, the times shown are the sum of the download (source) node, the GPU load node and the feature extraction node times. As we can see, the values are only slightly higher (around 50 milliseconds, at the highest) than is the case for the single-algorithm *feature extractor* time with persistence disabled (Figure 5.10). This means that there is very little performance penalty when extracting features from three algorithms, instead of only one. If we were to do the whole feature extraction process one algorithm at a time, the overall feature extraction time would amount to more than a second, while in this case it can be achieved in just under 400 milliseconds.

5.5.1 Pipeline steps comparison

Table 5.5 shows us the different times for the different *feature extractor* pipeline steps. Now, instead of presenting the time for each different algorithm, we present the overall time, as the *feature extractor* extracts features from all algorithms simultaneously.

We can see that, compared to Table 5.4, there is only a slight increase in the times. Especially the source node and the feature extraction nodes. However, that time increase is very light (tens of milliseconds), compared to the benefit gained from processing all

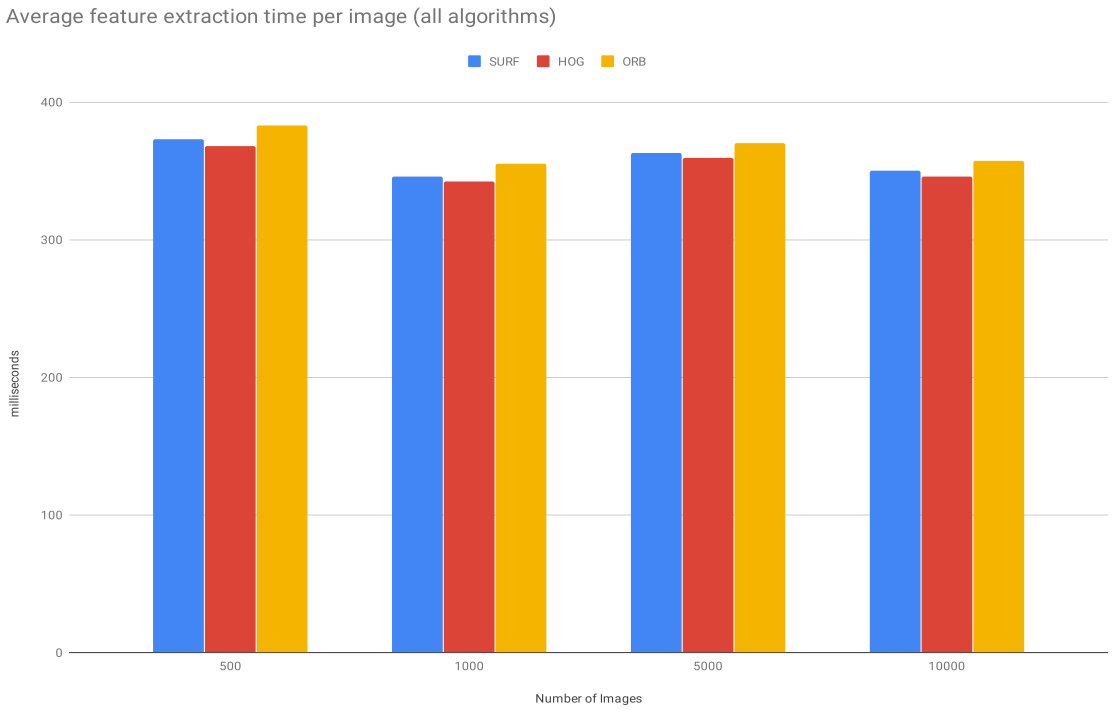

Figure 5.15: *Average feature extraction time (all algorithms enabled), per image.*

Table 5.5: Feature Extraction pipeline steps (all algorithms enabled) average execution time, in milliseconds.

	500 Images	**1000 Images**	**5000 Images**	**10000 Images**
Source Node	361	335	350	337
GPU Load Node	4	4	4	4
Image Date Load Node	0.043	0.043	0.045	0.05
SURF Feature Extraction Node	8	7	9	9
HOG Feature Extraction Node	3	3	5	5
ORB Feature Extraction Node	18	16	16	16
Image Date Comparison Node	0.011	0.010	0.005	0.005

algorithms at once. If we sum the overall time increase in the nodes, it will still be lower than the total time we would spend processing only one algorithm at a time, as we did in Subsection 5.4.1 and Subsection 5.4.3.

The increase in time here is, as before, due to CPU sharing. Now there are two more nodes than before executing feature extraction concurrently with the other pipeline nodes. This means that there are more nodes sharing the CPU and, thus, a slight delay is introduced. It is not as significant as when we enable the feature save node, because the nodes introduced are significantly faster at executing their computations and use much less processor time than the feature save node.

5.5.2 Summary

There is a clear advantage in extracting features from images using all algorithms at once, if the user requires it. There is a lot of time to be gained by doing the process in this manner, rather than one algorithm at a time. This is the way our system was designed and meant to be executed in. It is also the way we can take full advantage of the cluster at our disposal. Users with different cluster configurations may benefit from different configurations of our system, using more or less *feature extractors* and *feature matchers*. The good news is, our system was designed in a way that allows several configurations to execute it, it is scalable and adapts well to any form of cluster configuration, meaning different users, with different constraints and hardware can still take advantage of this system in the best way possible.

5.6 Comparison with CPU Implementation

In order to properly evaluate and be able to claim the efficiency of using the GPU over the CPU, we present, in this section, a comparison between the regular implementation of our library (the one that benefits from GPU-accelerated implementations of the algorithms used) and a CPU implementation of our library. In the latter, the GPU is not used at any point of the pipeline. That is the only difference between these two systems. In the CPU implementation, neither the *feature extractor* nor the *feature matcher* use the GPU in any way. This means that all feature extraction and feature matching algorithms execute entirely on the CPU.

Like before, we split the evaluation into two categories: feature extraction and feature matching. This is because they are very different processes, and are typically executed on different machines. This allows us to more finely evaluate the performance of each major role of our system, and properly investigate how much of a benefit can be obtained by executing our solution on the GPU.

All tests were run without any persistence functionalities enabled, as we already evaluated the performance penalty they entail in the execution of our system. Since these do not depend on the GPU in any way, we decided to turn them off. Regarding the configuration of our system, we went with the all algorithms enabled approach (similarly to Section 5.5), with one cluster machine performing feature extraction and three machines performing feature matching (*i.e.* one for each algorithm).

We decided not to test each algorithm individually (as in Section 5.4) because the only difference in this case is in regards to the performance of the *feature extractor*, where instead of extracting features from all algorithms simultaneously, features are extracted from only one algorithm. However, since our system was designed to extract features from multiple algorithms, this is the comparison we present in this section.

Because the CPU tests were executed under a different cluster state, and with a large time difference between the previously shown GPU tests, we decided to execute GPU tests

once again, for comparison purposes. These GPU tests are what will be shown in this section, and they will be the basis of comparison against the CPU tests. This will ensure a more rigorous comparison between CPU and GPU executions, as they are both executed on a shorter window of time and on more identical cluster and network conditions. If we were to simply show the CPU results and compare them with the GPU results shown in the previous sections, the results would be disparate and inaccurate, because the cluster conditions could have suffered some alterations since they were last executed.

5.6.1 Feature Extraction

Figure 5.16 shows us the comparison between execution times of the *feature extractor* for both the CPU and the GPU. It is very clear from the chart that executing the *feature extractor* on the GPU is much more time efficient than doing it on the CPU. This is quite an obvious and expected result, as the GPU implementations of the feature extraction algorithms entail in much faster extraction times.

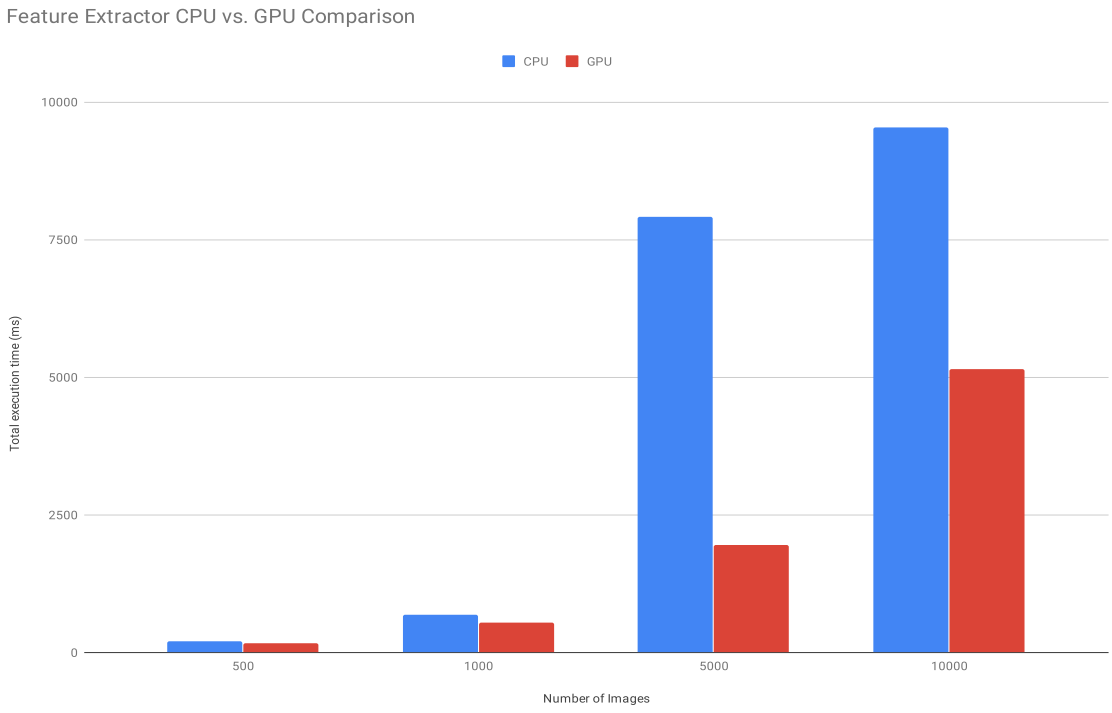

Figure 5.16: *Feature Extractor CPU vs. GPU execution time comparison*

For lower amounts of images, the difference is not so significant because, as we mentioned previously, when dealing with low amounts of images, the source node is too slow when compared to other pipeline nodes, which results in some starvation of the source node's successors. This means there is not much advantage to be gained in optimizing the execution time of the other pipeline nodes, because they will not be fed images fast enough for that difference to be noticeable. Of course, even though the difference is small, it is still more efficient to execute the *feature extractor* on the GPU.

When we move on to larger amounts of images (5000 and more), the time difference becomes quite apparent. Now, since the source node no longer introduces starvation, the feature extraction step is fed with images at a sufficient rate and it becomes clearly advantageous to execute feature extraction on the GPU, which is obviously much faster than executing it on the CPU.

Figure 5.17 presents the speedup that the GPU offers in the execution time of the *feature extractor*. We can clearly observe, as we stated before, that the speedup of the GPU over the CPU in regards to the execution time of the *feature extractor*, is quite low when dealing with low amounts of images (*i.e.* 1.16x speedup for 500 images and 1.28x speedup for 1000 images).

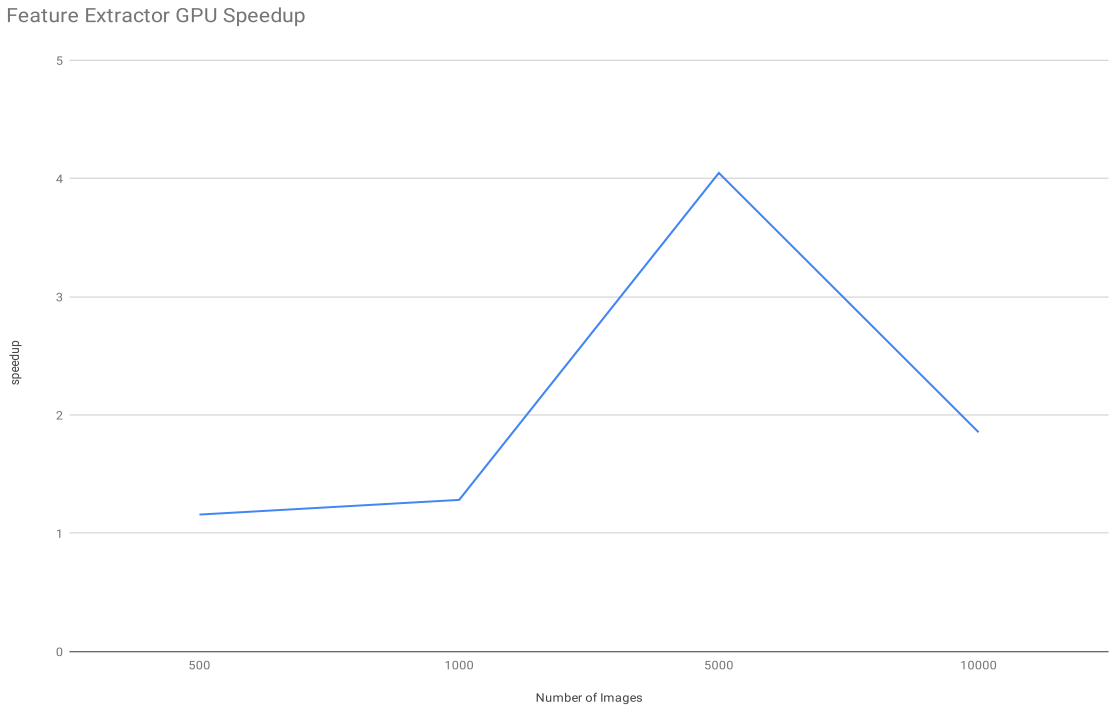

Figure 5.17: *Feature Extractor GPU speedup*

When processing 5000 images we see a peak in GPU performance. More than 4x speedup over CPU execution. This is very significant and clearly shows the power that the GPU can have when dealing with feature extraction algorithms. When we advance to 10000 images, however, the speedup falls back down to around 1.86x. This is due to the fact that, at that large amount of images, a lot more image comparisons need to be made, as we previously evaluated. This means that the overall pipeline is slowed down by the sheer amount of comparisons that need to be computed, and the *feature extractor* has to wait on the *feature matcher* to be ready to receive more image features (as we saw previously in Subsection 5.4.1). This slows down the execution time of the *feature extractor*, but does not necessarily make it slower at extracting features from images. In fact, when using the GPU, the *feature extractor* always completes its execution faster than

Table 5.6: Feature Extraction pipeline steps average execution time, in milliseconds, for CPU and GPU.

	500 Images		1000 Images		5000 Images		10000 Images	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
SURF FE Node	12	8	12	7	11	5	11	3
HOG FE Node	5	3	5	3	5	5	5	1
ORB FE Node	5	18	6	13	6	10	6	8

when using the CPU.

Finally, in order to more finely evaluate the impact of executing the *feature extractor* on the GPU, we need to take a closer look at the execution times of the feature extraction pipeline step of the *feature extractor*. There is no need to consider the times of source node or the image date nodes, because they are always executed on the CPU, regardless of the implementation of our system (GPU or CPU). We also do not require evaluating the GPU/CPU load node. This is because they take the exact same time in both versions, as the amount of work that needs to be done in that node is very similar in both cases.

This pipeline steps comparison can be observed in Table 5.6. The table only shows the times for the actual feature extraction nodes, as they are the only ones that are influenced by the GPU. None of the other nodes in the *feature extractor* role benefit from execution on the GPU.

As we can see from the table, the average time for the extraction of features from each image is always faster when using the GPU, with the exception of the ORB algorithm. However, the difference becomes smaller and smaller as we increase the total number of images processed. The performance penalty of executing the ORB feature extraction process on the GPU comes from several factors, such as the actual OpenCV implementation. However, the time difference is not so noticeable when dealing with higher amounts of images. Additionally, the overall execution time of the whole pipeline (including feature matching) is faster for ORB when executing on the GPU, as we will see subsequently, because the feature matching step will always be faster. This means that, even if features are extracted at a slower rate for ORB when using the GPU, the feature matching time on the GPU will make up significantly for that lost time. The rest of the algorithms show clear advantage when executing on the GPU (with the exception or HOG for 5000 images, where the times are the same).

5.6.2 Feature Matching

Despite executing our system in the all-algorithm configuration (1 *feature extractor* and 3 *feature matchers*), in this subsection we present the results of the performance of the *feature matcher* for each algorithm individually, to better be able to evaluate the impact of executing our solution using the CPU.

Figure 5.18 shows us a comparison of the total execution time, in milliseconds, for the *feature matcher* (using both the CPU and the GPU), when matching features extracted using the SURF algorithm. When considering any number of images, feature matching is always faster when executing on the GPU. The difference is not so significant with low amounts of images, but becomes very apparent when dealing with 5000 or more images. There is a clear advantage in performance when executing feature matching on the GPU. The advantage in this case is much more significant than in the case of the *feature extractor*, because the feature matching algorithms are highly parallelizable, composed by a very large number of small calculations, which is a task that GPUs excel at performing.

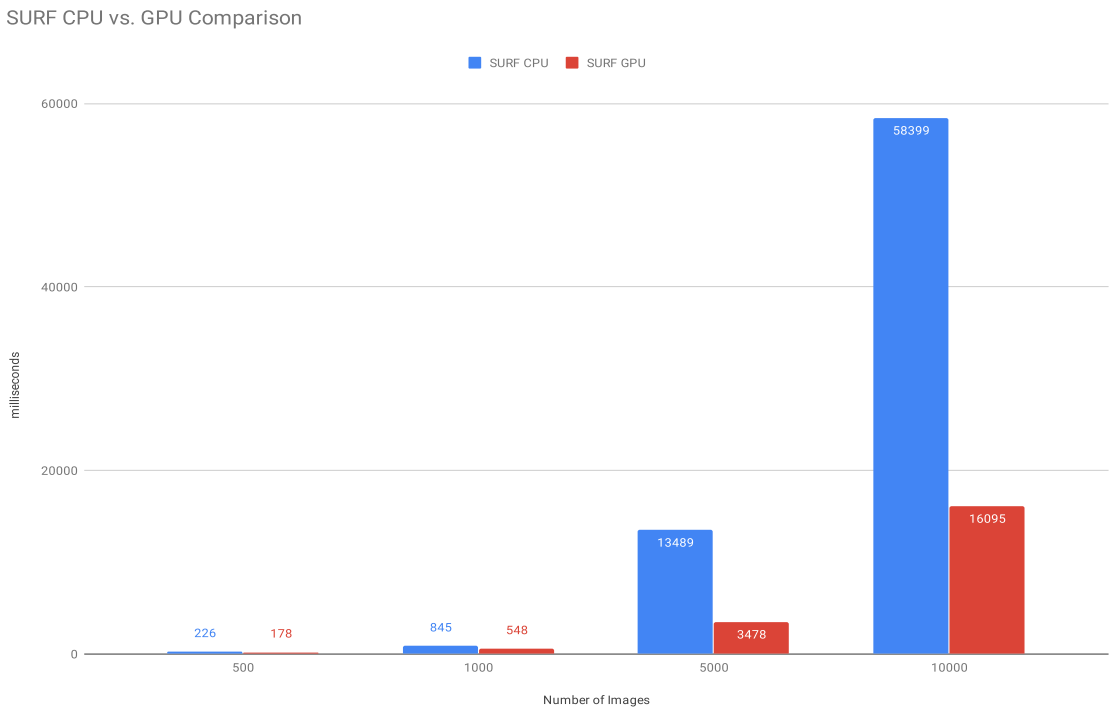

Figure 5.18: *SURF feature matcher CPU vs. GPU execution time comparison*

Figure 5.19 compares the execution time, in milliseconds, of the *feature matcher* when using features extracted using the ORB algorithm. The feature matching method in this case is very similar to the case of the SURF algorithm (brute-force k-nearest neighbours), only differing in the distance norm used (as explained in Section 3.1.2). As expected, similarly to SURF, the execution time of the *feature matcher* when matching ORB features is much lower when executing on the GPU.

There is not much to say about these values, as they follow a very similar trend to the SURF algorithm, displaying much better performance when executing on the GPU, mainly due to the similarity of the matching methods. As before, the performance difference becomes more and more apparent as we increase the total number of images processed. So far, it is easy to see why choosing to execute these algorithms on the GPU is a great choice.

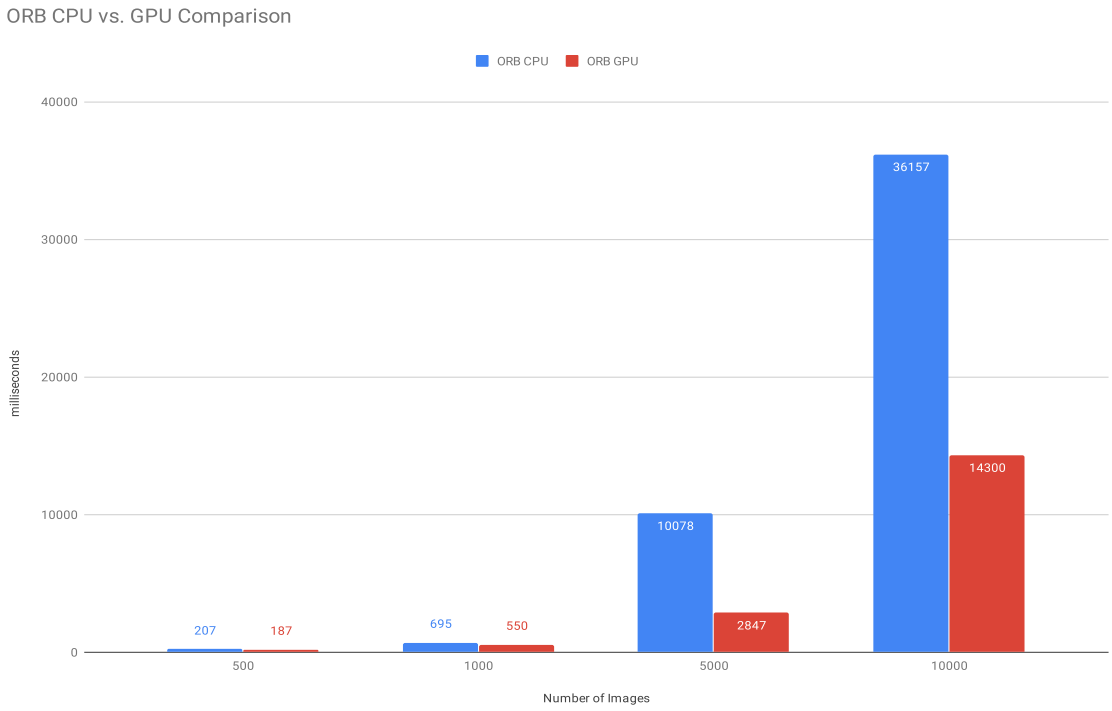

Figure 5.19: *ORB feature matcher CPU vs. GPU execution time comparison*

In order to complete our assessment of performance between executions on the CPU and executions on the GPU, we also need to evaluate the performance of the HOG algorithm. As we have learned from before, it is the slowest of all three algorithms at performing feature matching, even though its features are usually the fastest to extract. This comparison is shown in Figure 5.20.

The results here are quite different from before and may be unexpected. We can see a better performance for the GPU when dealing with very low amounts of images, but when we move on to 5000 or more images, the CPU starts to actually be faster at executing feature matching between features extracted using the HOG algorithm. In fact, when processing 10000 images, the CPU version of our system is close to 5x faster than the GPU version of our system.

The reason for this is the fact that HOG's feature matching process does not have a full GPU implementation. This means that, eventually, in the GPU version of our system, features will have to be downloaded from the GPU to the CPU in order to be matched. This is a process that takes time. That time will, eventually, slow down the overall solution significantly, because it will be repeated literally millions of times. Even though HOG feature extraction is faster on the GPU, that difference does not make up for the time lost on feature matching. A solution for this should be considered in future work. A better, fine-tuned GPU implementation of the histogram comparison process (Section 3.1.2) that performs the necessary calculations on the GPU, without relying on the CPU and, hence, without requiring the constant GPU-CPU communication costs and

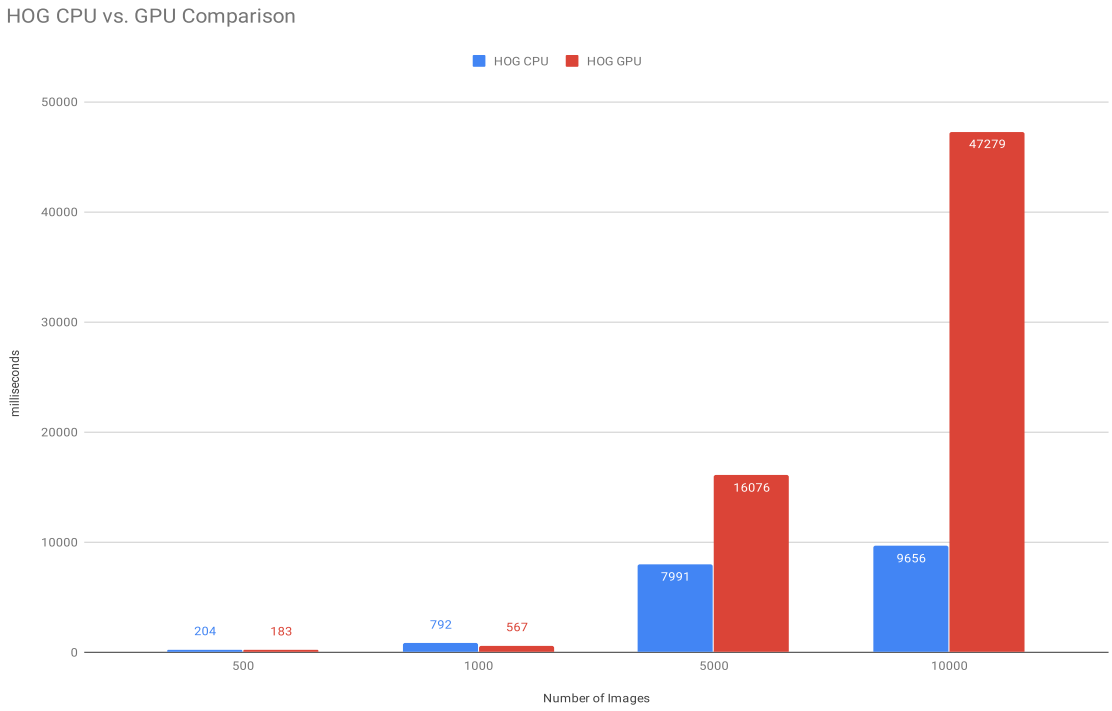

Figure 5.20: *HOG feature matcher CPU vs. GPU execution time comparison*

the resulting performance penalty. This would yield an overall better performance of our system when executed on the GPU.

This allows us to conclude that if we want to extract and match features using all algorithms and achieve the fastest performance possible of our system, we need to use the CPU version for the HOG algorithm, while using the GPU version of the SURF and ORB algorithms.

The last chart, present in Figure 5.21, shows us a comparison between the speedup in the execution time offered by execution on the GPU vs. execution on the CPU. As we can see, for both SURF and ORB, with any amount of images, there is a clear speedup in the execution time of the *feature matcher* when using the GPU.

As expected, the speedup is not very significant when processing small amounts of images, as the amount of comparisons that are made is very small, which means the CPU can easily keep up with the necessary computations. The benefit of using the GPU here, although present, is not as apparent. The benefit of the GPU only becomes very apparent when processing larger amounts of images, because large amounts of images entails in a very large amount of image comparisons and, as we have explored previously, GPUs excel at performing very large amounts of small computations simultaneously while CPUs, although they can typically perform bigger, more intensive computations better, they can not compete with the level of parallelization that the GPU offers. So, as expected, the speedup for the SURF and ORB algorithms peaks at 5000 images and slowly lowers around 10000 images, due to the significantly higher amount of image comparisons that

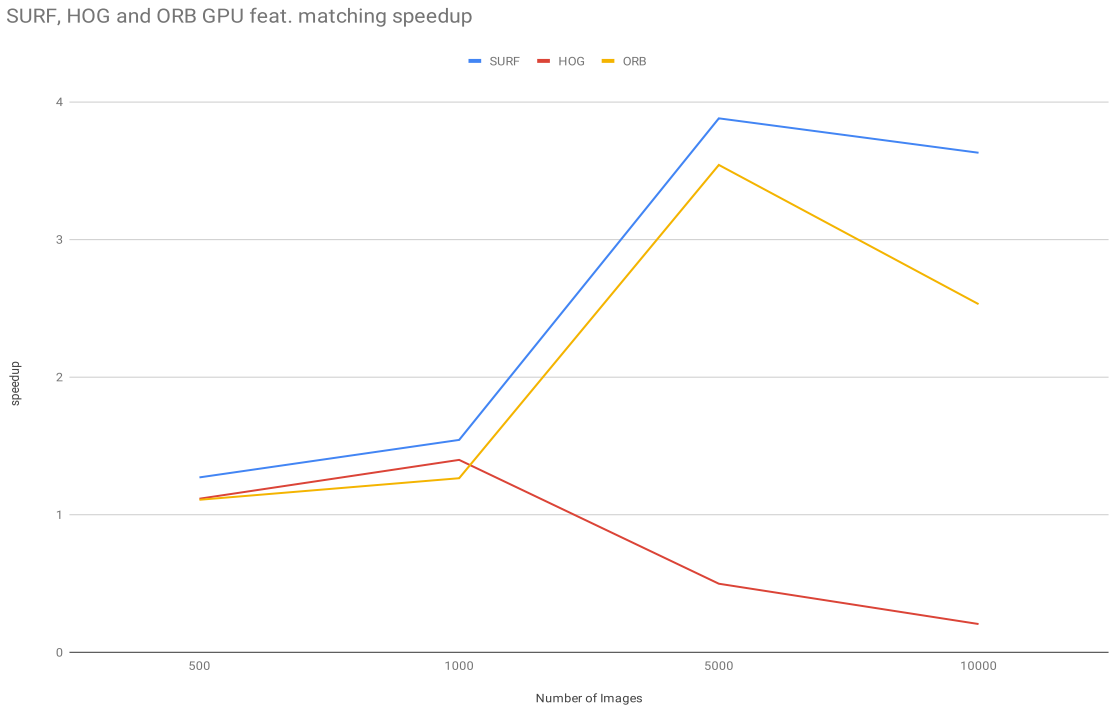

Figure 5.21: *All algorithms feature matching speedup*

need to be made. In the case of 5000 images, the number of comparisons that need to be computed is much smaller than in the case of 10000 images, which means that the GPU can easily perform this task, and no delays are introduced in the pipeline. It is sort of a "sweet spot"of performance. When moving on the 10000 images, the very large number of comparisons that need to be made will introduce more delays in the system, as we have explored in the previous sections, which also means the GPU's performance will be slightly slowed down. Bear in mind that, although slower when processing 10000 images, the GPU is still more than 3.5x faster than the CPU for the matching of SURF features and more than 2.5x faster for ORB feature matching. The benefit of using the GPU in these cases is quite evident.

For the HOG algorithm, however, the speedup falls below 1 when processing 5000 or more images, meaning it is actually slower to execute the *feature matcher* on the GPU than executing it on the CPU, as we have explained previously.

In order to obtain a more detailed view on the performance of the GPU over the CPU, we present a table comparing the average CPU and GPU execution time for the feature matching pipeline step of the *feature matcher* role, as we did in the previous subsection. Table 5.7 details these values.

As we can see, and as expected, the average feature matching times are always lower in the GPU in the cases of the SURF and ORB algorithms. There is, once again, a clear performance benefit in executing these steps using the GPU. We can also note the delay of executing feature matching between HOG features, starting from 1000 images. The delay

Table 5.7: Feature Matching pipeline steps average execution time, in milliseconds, for CPU and GPU.

	500 Images		1000 Images		5000 Images		10000 Images	
	CPU	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
SURF FM Node	11	1	15	1	12	2	15	3
HOG FM Node	1	1	1	8	1	15	1	12
ORB FM Node	1	1	6	1	8	2	9	3

in this case is very significant, and showcases how much of a performance penalty we get in the case of GPU HOG feature matching. This is the time we mentioned previously, resulting from the CPU-GPU communication that is necessary before matching HOG features.

5.6.3 Summary

This section allowed us to obtain a detailed view of the benefits of using the GPU for many of the algorithms our system uses. In general, the GPU performs the necessary computations much faster than the CPU, and results in an overall faster execution time of the whole pipeline.

Now, we can empirically claim that the GPU is, in fact, very beneficial for these types of algorithms and for our system. These tests also allowed us to benchmark our library using the CPU and resulted in the conclusion that the HOG algorithm actually has worse performance when executed on the GPU, due to OpenCV's implementation of the histogram comparison method and the necessity of downloading image features from the GPU to the CPU before performing the matching computation. This means that, in order to achieve the best possible performance of our system (while using all algorithms simultaneously), a user should opt for a configuration that uses the GPU for SURF and ORB feature extraction and matching and the CPU for HOG feature extraction and matching. This is the configuration that allows the lowest execution time of the whole pipeline, when considering the completion of all algorithms.

Still, it is quite evident from this section that the GPU offers tremendous benefits to our system, as it speeds up the execution of the *feature extractor* and the *feature matcher* up to 4 times. Additionally, future work focusing on the improvement of the HOG feature matching algorithm, through implementation of a GPU version of the histogram comparison method and the elimination of the undesired CPU-GPU communication costs, would result in all-around faster performance for our system, and should be considered.

5.7 Query Tests and Results

The times that a query takes to process and complete are completely independent of the algorithm used to calculate the similarity score. This is due to the fact that, in order to

execute a query, all the system needs is the graph containing image identifiers and scores. Since the scores are normalized for all algorithms, there is no difference between loading a SURF, HOG or ORB graph (in Gunrock's format), processing it and outputting a query. Queries do not depend, in any way, from the type of features extracted, or even the type of matching method used, for that matter. They only care about the values that are present in the final version of the graph, which are merely integer numbers.

A query has essentially two steps that take time:

- Converting the SS Graph object to a format compatible with Gunrock
- Calling Gunrock's primitive and obtaining the results

These are the two steps we need to measure in order to properly evaluate the performance of a query. They both depend exclusively on the number of nodes (images) and edges (image comparisons) in the graph. And these, in turn, depend only on the total number of images processed. Hence we present the results, averaged for all tests made in this Chapter, for each number of images tested.

Figure 5.22 shows us the time, in milliseconds, that Gunrock spends converting the graph presented in our format (SS Graph) to a format it can process. This is a process that is always required before the execution of one or several queries. The time taken for this process to execute increases exponentially with the amount of images processed, which is due to the fact that as we increase the number of images we process, we exponentially increase the number of comparisons our system makes. Hence, the graph will contain exponentially more edges, as the number of nodes increases. As we saw before, a graph of 500 nodes (images) has 124750 edges (image comparisons), while a graph of 10000 nodes contains 49995000 edges.

Since this process only needs to be executed once before running queries on the graph, it is wise to execute it before running several queries. Once this process is complete, a user can execute as many queries as he would like. It is even possible to change the type of query made (*i.e.* changing the Gunrock graph primitive) without having to run the conversion process again. Still, 60s to process a graph with almost 50 million edges and 10 thousand nodes is not a big price to pay, considering the amount of image similarity information present on that graph.

Figure 5.23 presents the time, in milliseconds, that Gunrock spends executing the actual query computation. It is essentially the Single-source shortest path (Dijkstra's algorithm) runtime for the given source image (or images). It outputs the lowest scores to the images that the source image is connected to via an edge which means, in other words, it outputs the images to which the source image is most similar to. As we can see, this process, executed on the GPU, is extremely fast. Even for 10 thousand images and close to 50 million nodes, it only takes Gunrock a little more than 15 milliseconds to execute the query.

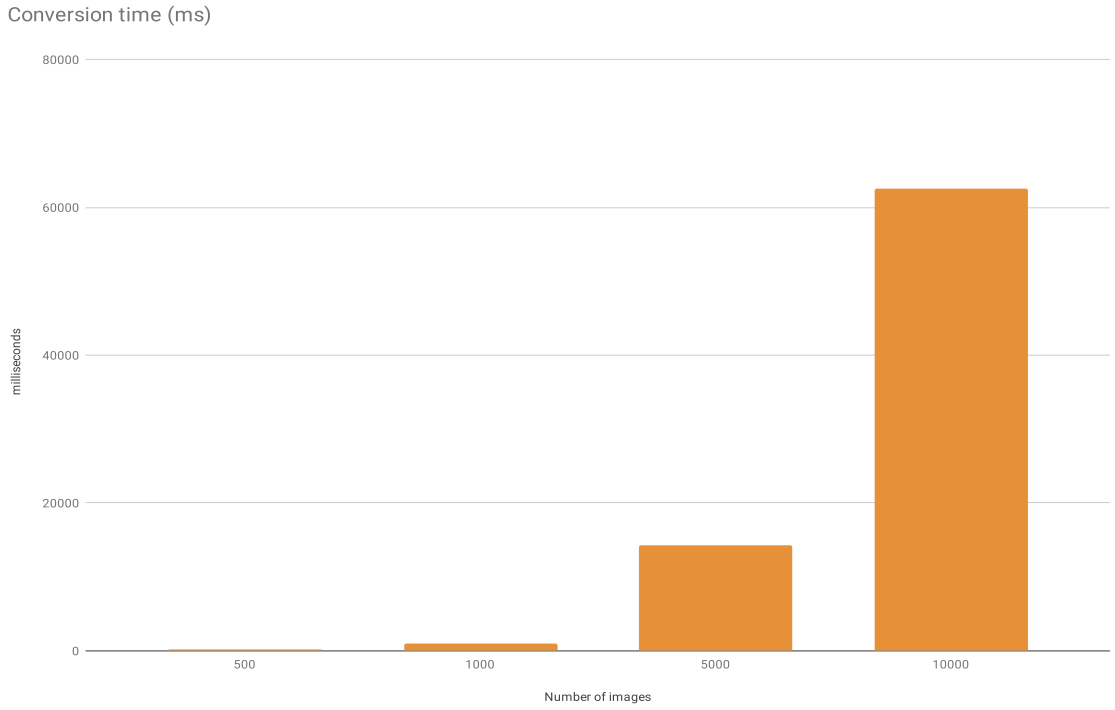

Figure 5.22: *Query graph conversion time, in milliseconds.*

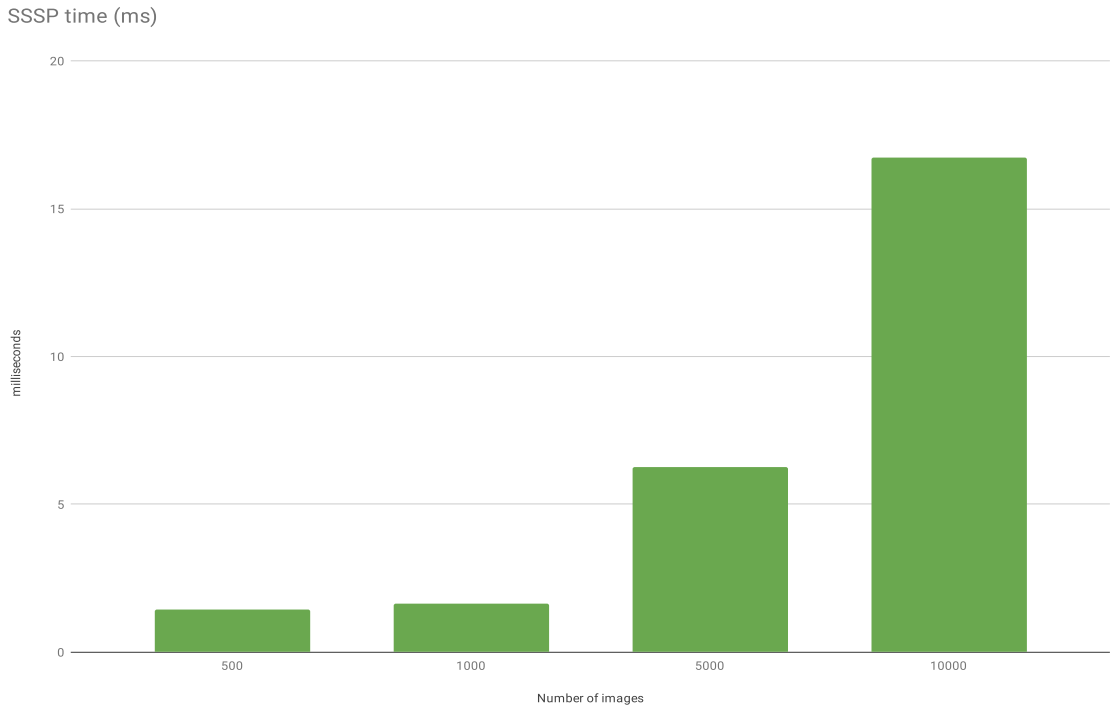

Figure 5.23: *Query primitive execution time, in milliseconds.*

Thus, we can safely conclude that the biggest bottleneck in the execution of a query is the time it takes for Gunrock to convert the graph we feed it to a format it can process. It

consists, essentially, of uploading the graph information to the GPU so it can be processed by the computation that executes the query. The time it takes to execute a query is easily negligible when compared to the rest of the pipeline of the system (both image processing and query processing).

This is, however, good news, because the focus of our system is to enable the quick identification of image similarity. If we consider a possible application, that contains the graph in Gunrock's format at any time, ready to process, it can output similarity queries in fractions of a second. It is easy to imagine an application that has, at any given time, a library of ten thousand images all compared and ready to query. Although the processing of those ten thousand images took a few short hours (depending on the algorithms used), once it is complete, there is no further need to run it. The only computation that needs to be made after that is the computation of queries which, as we have seen, is extremely fast and efficient.

5.7.1 Query precision evaluation

Although we have tested how a query performs in terms of execution time, it is also important to evaluate how precise it is. Although we can not assign an empirical value to the precision, we can informally compare the images the query states as the most similar, for each algorithm, and debate whether or not there is some degree of similarity. It is also important to note that there will always be false matches and, just because a similarity query outputs a certain image as the most similar, it merely means it is the most similar image of the images currently present in library. It may just be the case that the two images shown are in fact the most similar among the ones available, as the library is not complete enough for an obvious, close visual match.

In any case, we did 10 similarity queries using the first ten images as source images, for all three algorithms, in a pool of 2000 images. ORB did not obtain any match worth mentioning, with scores always above 97. SURF obtained a couple of matches we though were worth mentioning, such as the ones present in Figure 5.24 and Figure 5.25.

In the first pair of images, there is a similarity in the people present, as well as the visual similarity between the camera tripod (image on the right) and the microphone tripods (image on the left). In the second pair, there is a visual similarity between the bouquet the woman is holding, as well as the jar of flowers behind her (left image) and the flower (right image). There is also a visual similarity between the white dress and its contours and the plants white color and contours. As SURF is especially adept at object detection, it finds image similarities such as these ones.

HOG also finds some similarities we though were worth mentioning.

Figure 5.26 shows the first match. The two image are clearly visually similar and both contain plates of food. The second pair, in Figure 5.27, shows us the difference between the SURF and HOG algorithms. As we can see, SURF found a completely different match for the same image (shown in Figure 5.24) to what HOG found. While SURF withdrew

Figure 5.24: *Query result example 1, using SURF features.*

Figure 5.25: *Query result example 2, using SURF features.*

the similarity from the objects present in both images, as we previously saw, HOG found the image on the right in Figure 5.27 to be the most similar due to the presence of human shapes in both of them. Lastly, in Figure 5.28, HOG found a similarity due to the curtains present in the background of both images. HOG also excels at finding patterns and gradients in the background, which is what these two images have in common.

These are only a few examples of visually similar matches that our system presents. They intend to show the reader the effectiveness of the system at finding visually similar images, and the possible advantage that can be gained in using multiple algorithms (such as the case with Figure 5.24 and Figure 5.27). We may have different needs in any particular query, which means there may be algorithms more suited towards the type of query we wish to execute. Enabling all of those algorithms, as we do, is a powerful tool for users and applications.

Figure 5.26: *Query result example 1, using HOG features.*

Figure 5.27: *Query result example 2, using HOG features.*

Figure 5.28: *Query result example 3, using HOG features.*

5.7.1.1 N-most Similar Queries

In order to further evaluate the precision of queries, we also executed a different type of query in our system. The n-most similar images query. This query finds the n most similar images to a given source image, instead of only the most similar. In order to demonstrate this functionality and further evaluate the precision of our library, we present the 20 most similar images for each of the three algorithms. In this case, we execute the query in a pool of 10000 images, finding, for each algorithm (SURF, HOG and ORB), the 20 most similar images to a given source image.

It is important to bear in mind that the images shown are merely the 20 most similar to the source image in the pool of images that are available. It does not mean they are all very visually similar, it merely means they are the most similar amongst the ones that were processed.

The first query, corresponding to SURF features, can be observed in Figure 5.29. The source image corresponds to the image at the top of the figure. We can see some form of resemblance amongst some images, such as an industrial setting and sharp edges and corners (corresponding to buildings, machines, furniture or even billboards). These are likely resulting from SURF matches from the source image key points (building corners, vehicle corners) to other image key points (*i.e.* other building corners, billboard corners). There are, of course, some images that bare no visual resemblance to the source image but this is due to the fact that the image pool is not nearly complete enough to yield 20 precise, visually similar matches. The images shown are merely the 20 most similar amongst the 10000 processed.

Figure 5.30 shows the query for ORB features. The source image is the same as before. Like before, some images bare visual resemblance, such as the ones containing buildings and industrial structures, as well as vehicles. Comparing this figure with Figure 5.29 also showcases the difference between what SURF considers to be key points and between what ORB considers to be key points. It also illustrates that using a different algorithm results in completely different matches, as we have explored previously. While some images are quite visually similar, others are not so much. As before, this is due to the fact that the images shown are the 20 most similar amongst the pool of images processed, meaning that the other 9980 images are not as similar to the ones shown, according to the score metric we designed.

Finally, in Figure 5.31, we present the 20 most similar images resulting from executing the query with HOG features. The resulting images show several visual similarities, like other buildings and structures, such as the ones present in the source image. Most resulting images are images of city skylines, buildings and structures. Considering this result, we can state that HOG is the algorithm (out of all three) which produces matches that are closer to what we would consider visually similar. Still, as we previously saw, it is important to consider that other algorithms have different use cases and situations where they excel at. If we consider all three figures (Figure 5.29, Figure 5.30 and Figure 5.31), we can clearly observe that features extracted using different algorithms result in very different matches. It is a matter of deciding which algorithm suits our needs better. The advantage of being able to extract features from all three (and possibly more, considering the easy expandability of our system), and perform queries using any of the algorithms used, is a great functionality of our system, that can be taken advantage of in order to suit any use case.

5.8 Summary

After these experimental evaluations, we can safely assume that our system is able to compare thousands of images every second. It achieves its goal of being a fast image comparison, similarity search system. Besides fast image comparison, the system also features very fast query execution, a major necessity, since queries are the main focus of our program. It suffers on the image processing side as it can not process many images simultaneously, due to memory issues. It also performs many comparisons that may be unnecessary. Future work that addresses these two issues, *i.e.* memory and reduction in the number similarity computations (search-space reduction) will yield a very powerful image processing library. However, as it stands, it is still quite useful. Although the image processing step is slow (feature extraction + feature matching), its output is incredibly powerful and useful. Once that output information is accessible, the comparison of images and similarity search computation is extremely fast, and has several use cases.

We also evaluated that the enabling of the feature save node introduces several performance issues in the whole pipeline, slowing down the process in almost all pipeline nodes. The enabling of this feature needs to be considered carefully, as it will heavily penalize performance.

We assessed that CPU sharing is an issue, and the number of pipeline nodes and their performance needs to be controlled carefully, as to not introduce performance issues in the rest of the nodes. We also verified that the source node is the most computationally intensive step of the whole *feature extractor* pipeline, but it can be improved by caching images in the machine, or using pre-downloaded images instead of images provided by a stream. Future work should also include the optimization of certain pipeline nodes, such as the source node (particularly its image download portion).

We also presented several experimental results that showed that executing the necessary algorithms using a GPU results in much better performance than would be achievable on a CPU, with the exception of the HOG algorithm for higher amounts of images. This is due to the fact that the GPU feature matching method for HOG features is not fully implemented in OpenCV. Future work should also focus on the optimization of this feature matching method on the GPU, designing a fast implementation in order to speed up this process and, thus, be able to take full advantage of the GPU when using our system.

In the next Chapter, we withdraw several more conclusions from these tests and the previous Chapters. We make an overview of the implementations and performance of our system, as well as its contributions. We also suggest future work needs for our system, and what problems they should focus on.

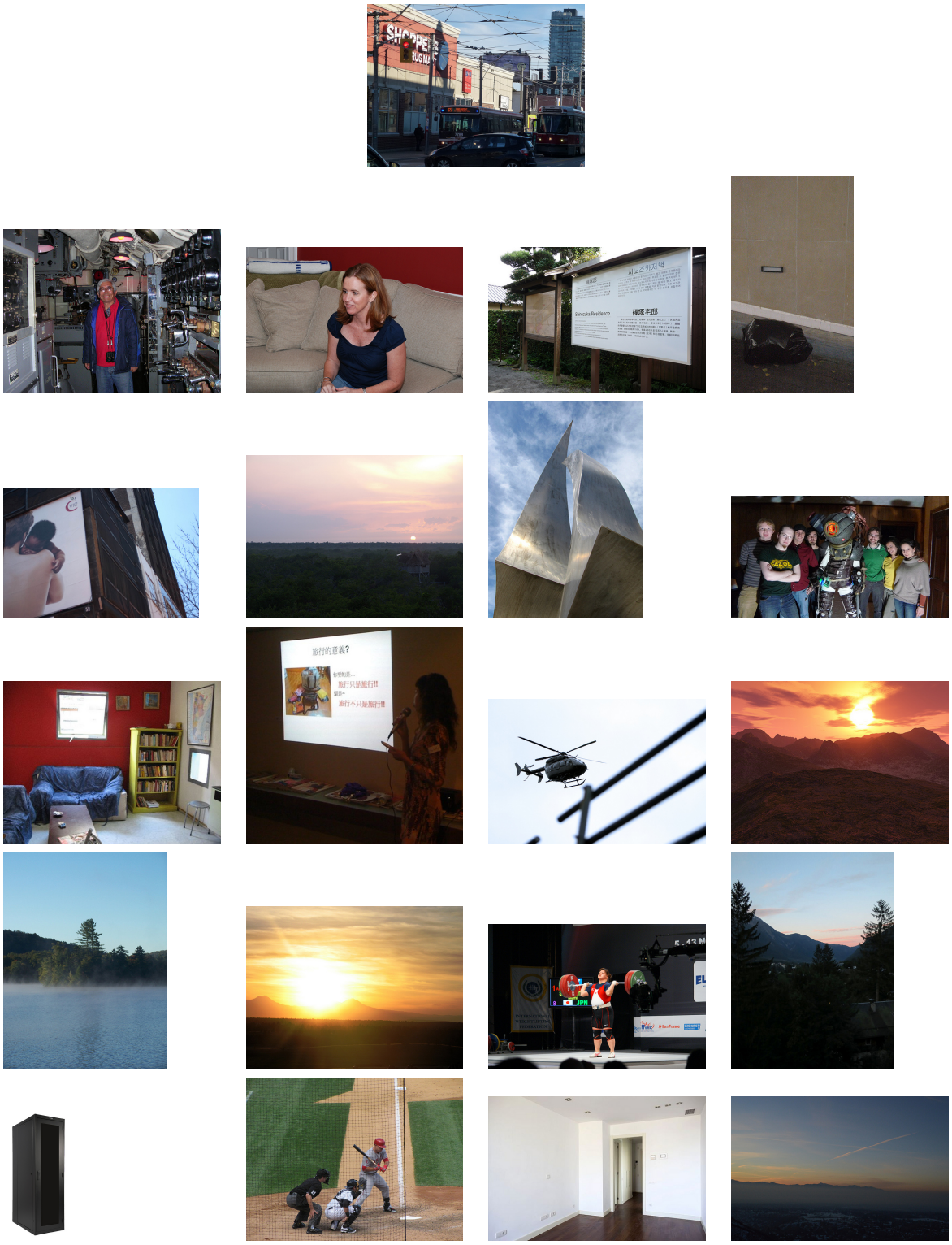

Figure 5.29: *20 most similar images query, using SURF features. The image at the top corresponds to the source image.*

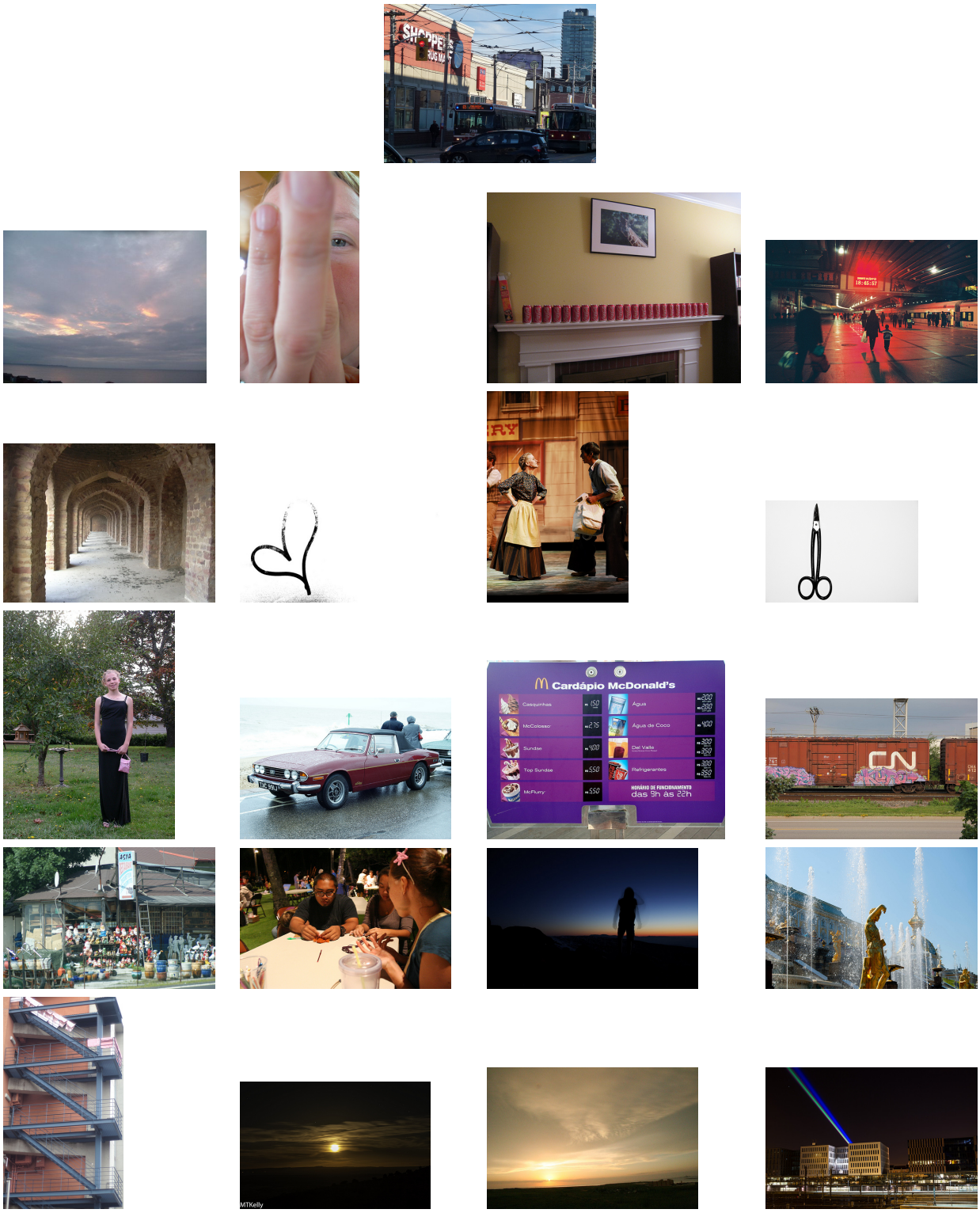

Figure 5.30: *20 most similar images query, using ORB features. The image at the top corresponds to the source image.*

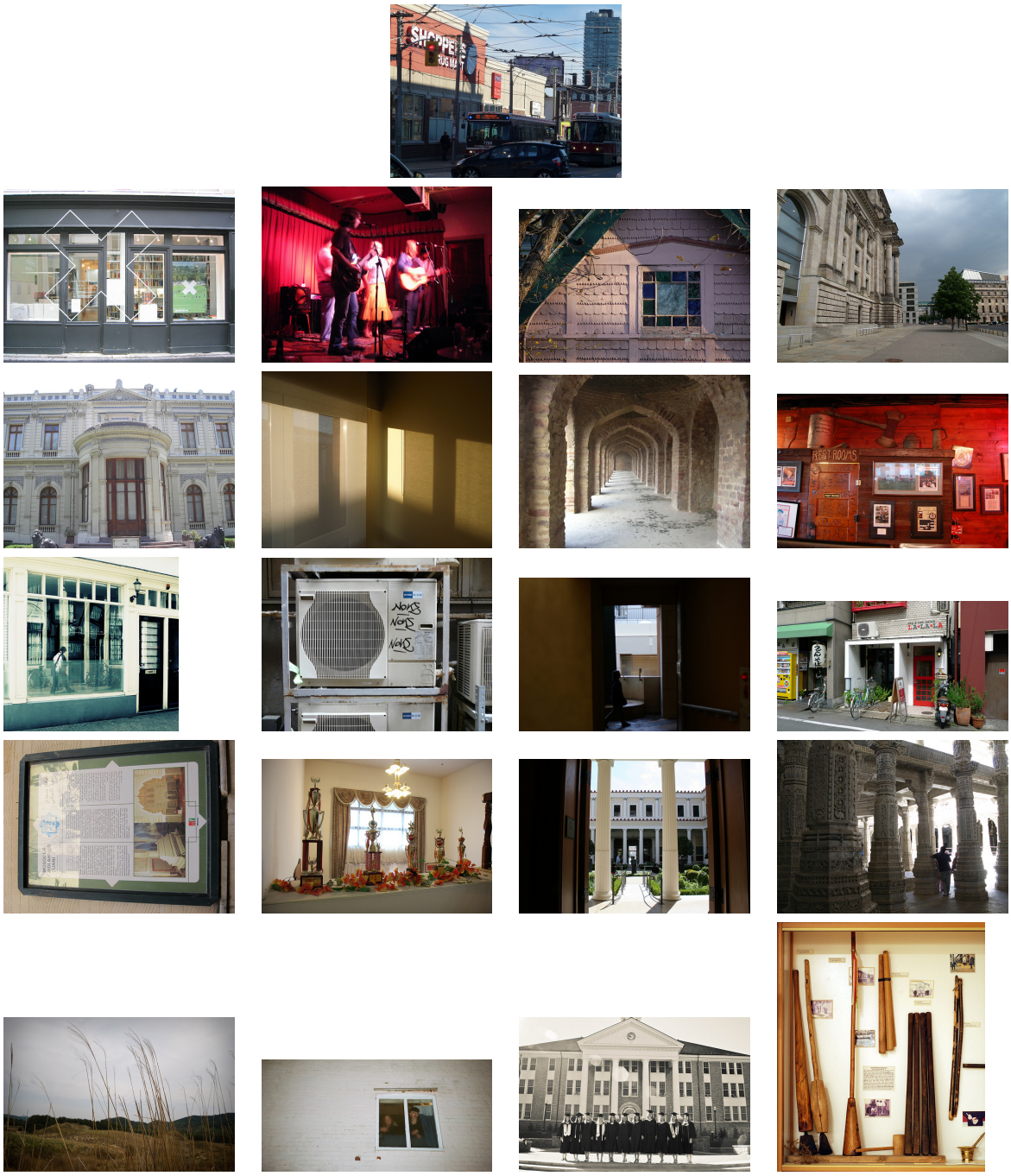

Figure 5.31: *20 most similar images query, using HOG features. The image at the top corresponds to the source image.*

Chapter 6

Conclusion and Future Work

This Chapter presents the conclusions of our work. It shows the work accomplished, whether or not our objectives were achieved, the status of our solution and what it allows, and the contributions it makes towards the state of the art. It also presents suggestions of future work, investigation opportunities opened by this thesis and possible improvements that could be made.

6.1 Conclusion

The final system we implemented represents a high-performance similarity search program. It is capable of performing thousands of image comparisons every second, and output similarity search results within milliseconds. It is capable of taking full advantage of a distributed, GPU-equipped cluster of machines in order to achieve the best performance possible. It is also very modular, allowing the interchangeability of several components (such as stream sources, algorithms, feature matching methods and query methods). The system is scalable and can take full advantage of the underlying hardware, as it is possible to execute it in several different configurations, adapting to the machines that execute it. The system uses fine-grained parallellism to take advantage of each machine vertically and uses an efficient distribution method, in order to spread out the computations it needs to perform through several machines, taking advantage of clusters in a horizontal fashion.

Our system outputs similarity graphs, containing diverse information about images, such as their similarity scores and their dates (and how far apart the dates are), and allow the easy computation of queries. The outputted graphs have several use cases, such as performing different types of queries and its compatibility with Gunrock, enabling the

execution of several different graph primitives. They essentially contain all the information our program computes regarding images, and can be used by applications in order to provide users with the ability to extract a multitude of information regarding images, especially image similarity, in a very fast manner.

It advances the state of the art by creating a different sort of system, that not only takes advantage of GPUs, but also takes advantage of distributed computing. Unlike other systems, it also enables the computation of image features in several different computer vision algorithms, as well as the execution of feature matching using one of several feature matching techniques. Usually, systems like these specialize in one computer vision algorithm and matching technique, whereas we provide multiple.

We can confidently state that we achieved our goal of designing a scalable, stream processing library that enables extremely fast and efficient image similarity search, being able to query a library of 10000 images and close to 50 million image scores in tens of milliseconds, as well as perform nearly five thousand image comparisons every second (for the fastest algorithms). The focus of this thesis was to take advantage of distributed computing and GPUs in order to speed up the process of enabling and executing similarity search queries on a library of thousands of images, and we can claim this goal was achieved. There are still several processes that need to be improved, such as the image processing steps. The image processing portion of our library is still in need of work, such as reducing the number of computations made (search-space reduction, which will significantly speed up this process) and reducing memory usage (enabling the processing of many more images).

The system is far from complete and this thesis was merely the first work on this subject. There are still several issues that need to be addressed in order to make this a much more complete, powerful library. Which is what we will explore in the upcoming section.

6.2 Future work

There are several possible improvements that could be made to our solution. One of which is the asynchronous writing of information to files (in the persistence functionalities, such as feature saving). This way, we could write to a buffer which was then read by another process that periodically writes that information to a file, without impacting performance. This would be a much better approach to our persistence functionalities, and should be considered in the future. Another issue we could improve is the way memory is handled throughout the program. As we increase the number of images we process, memory becomes an increasingly scarcer resource. There are several optimizations that could be made in order to improve this and, thus, increase the total number of images the system can process.

Future work should focus on problems such as these. Our work also enabled several research opportunities, that would further benefit the system's performance and scale.

First of which is the speeding up of the feature matching portion of the system, by reducing the number of image comparisons that are made, through search-space reduction techniques. Another factor that could be improved, regarding image comparisons, is the elimination of redundant and useless comparisons. Achieving this would significantly improve the performance of the system to very high levels. Because the feature matching step is the current bottleneck of our solution, it should be the first step that is focused on, by solving issues such as these. Besides, not executing comparisons between every single image that the system processes would also help significantly with memory issues. Because of this all-to-all comparison method, there is a need to keep every image feature descriptor in memory until all comparisons are made. This obviously harms memory usage significantly, and reducing the number of comparisons made would help with these memory constraints.

Usage of dynamic graphs stored in the GPU should also be considered. At the moment, our SS Graph datastructure is stored on RAM while it is being constructed and, at the moment of a query, it is processed and loaded to the GPU. This is a process that, as we saw, takes tens of seconds to execute. If our graph datastructure (or any another, for that matter) were to be stored in the GPU and dynamically modified, there would be no need for this conversion step.

Finally, the usage of multiple GPUs in a single machine should also be considered. Adapting the system so it takes advantage of more than one GPU would even further increase the way we take advantage of each machine vertically. This would significantly improve performance in all senses, and would mean we could also take advantage of clusters composed of machines equipped with more than one GPU.

Bibliography

[1] H. Bay, T. Tuytelaars, and L. Van Gool. "SURF: Speeded Up Robust Features". In: *Computer Vision – ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006. Proceedings, Part I*. Ed. by A. Leonardis, H. Bischof, and A. Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417. ISBN: 978-3-540-33833-8. DOI: 10.1007/11744023_32. URL: https://doi.org/10.1007/11744023_32.

[2] Boost. URL: https://www.boost.org/.

[3] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[4] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. "Brief: Binary robust independent elementary features". In: *European conference on computer vision*. Springer. 2010, pp. 778–792.

[5] G. Chen, Y. Ding, and X. Shen. "Sweet KNN: An Efficient KNN on GPU through Reconciliation between Redundancy Removal and Regularity". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 2017, pp. 621–632. DOI: 10.1109/ICDE.2017.116.

[6] I. Corporation. URL: https://www.threadingbuildingblocks.org/.

[7] Curl. URL: https://curl.haxx.se/.

[8] N. Dalal and B. Triggs. "Histograms of oriented gradients for human detection". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 1. 2005, 886–893 vol. 1. DOI: 10.1109/CVPR.2005.177.

[9] P. Ferrão, H. Marques, and H. Paulino. "Stream Processing on Hybrid CPU/Intel® Xeon Phi™, Systems". In: *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*. Vol. 11014. Lecture Notes in Computer Science. Springer, 2018, pp. 796–810. DOI: 10.1007/978-3-319-96983-1_56. URL: https://doi.org/10.1007/978-3-319-96983-1_56.

[10] A. S. Foundation. URL: http://storm.apache.org/index.html.

[11] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching". In: *2010 IEEE International Conference on Image Processing*. 2010, pp. 3757–3760. DOI: 10.1109/ICIP.2010.5654017.

[12] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel. "Buffer kd trees: processing massive nearest neighbor queries on GPUs". In: *International Conference on Machine Learning*. 2014, pp. 172–180.

[13] Google. URL: https://developers.google.com/protocol-buffers/.

[14] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. "Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores". In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. Munich, Germany: ACM, 2014, pp. 413–423. ISBN: 978-1-4503-2840-1. DOI: 10.1145/2591635.2667189. URL: http://doi.acm.org/10.1145/2591635.2667189.

[15] Q. Huang, P. Ang, P. Knowles, T. Nykiel, I. Tverdokhlib, A. Yajurvedi, P. Dapolito IV, X. Yan, M. Bykov, C. Liang, M. Talwar, A. Mathur, S. Kulkarni, M. Burke, and W. Lloyd. "SVE: Distributed Video Processing at Facebook Scale". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017, pp. 87–103. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132775. URL: http://doi.acm.org/10.1145/3132747.3132775.

[16] Intel. URL: https://www.threadingbuildingblocks.org/tutorial-intel-tbb-flow-graph.

[17] J. Johnson, M. Douze, and H. Jégou. "Billion-scale similarity search with GPUs". In: *CoRR* abs/1702.08734 (2017). arXiv: 1702.08734. URL: http://arxiv.org/abs/1702.08734.

[18] H. Jégou, M. Douze, C. Schmid, and P. Pérez. "Aggregating local descriptors into a compact image representation". In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2010, pp. 3304–3311. DOI: 10.1109/CVPR.2010.5540039.

[19] N. Leischner, V. Osipov, and P. Sanders. *Fermi architecture white paper*. 2009.

[20] D. G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *Int. J. Comput. Vision* 60.2 (2004), pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: https://doi.org/10.1023/B:VISI.0000029664.99615.94.

[21] D. G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60.2 (2004), pp. 91–110. ISSN: 1573-1405. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: https://doi.org/10.1023/B:VISI.0000029664.99615.94.

[22] S. Mallick. *Histogram of Oriented Gradients*. 2016. URL: https://www.learnopencv.com/histogram-of-oriented-gradients/.

[23] R. Marques, H. Paulino, F. Alexandre, and P. D. Medeiros. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Euro-Par 2013, Aachen, Germany, August 26-30, 2013. Proceedings*. Ed. by D. a. M. Felix Wolf Bernd Mohr. Lecture Notes in Computer Science 8097. Springer-Verlag. Aachen, Germany: Springer-Verlag, Aug. 2013, pp. 874–885. URL: http://link.springer.com/chapter/10.1007/978-3-642-40047-6_86.

[24] NVIDIA. URL: https://developer.nvidia.com/nvgraph.

[25] C. Nvidia. "Compute unified device architecture programming guide". In: (2007).

[26] J. Pan and D. Manocha. "Fast GPU-based locality sensitive hashing for k-nearest neighbor computation". In: *GIS*. 2011.

[27] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens. "Multi-GPU Graph Analytics". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 479–490. DOI: 10.1109/IPDPS.2017.117.

[28] E. Rosten and T. Drummond. "Machine Learning for High-speed Corner Detection". In: *Proceedings of the 9th European Conference on Computer Vision - Volume Part I*. ECCV'06. Graz, Austria: Springer-Verlag, 2006, pp. 430–443. ISBN: 3-540-33832-2, 978-3-540-33832-1. DOI: 10.1007/11744023_34. URL: http://dx.doi.org/10.1007/11744023_34.

[29] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. "ORB: An efficient alternative to SIFT or SURF". In: *Computer Vision (ICCV), 2011 IEEE international conference on*. IEEE. 2011, pp. 2564–2571.

[30] J. Shun and G. E. Blelloch. "Ligra: A Lightweight Graph Processing Framework for Shared Memory". In: *SIGPLAN Not.* 48.8 (Feb. 2013), pp. 135–146. ISSN: 0362-1340. DOI: 10.1145/2517327.2442530. URL: http://doi.acm.org/10.1145/2517327.2442530.

[31] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[32] N. I. of Standards and Technology. URL: https://math.nist.gov/MatrixMarket/.

[33] G. Teodoro, T. Pan, T. M. Kurc, J. Kong, L. A. D. Cooper, N. Podhorszki, S. Klasky, and J. H. Saltz. "High-throughput Analysis of Large Microscopy Image Datasets on CPU-GPU Cluster Platforms". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013, pp. 103–114. DOI: 10.1109/IPDPS.2013.11.

[34] N. TESLA. "P100 GPU Accelerator". In: *NVIDIA, Oct* (2016).

[35] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li. "YFCC100M: The New Data in Multimedia Research". In: *Commun. ACM* 59.2 (Jan. 2016), pp. 64–73. ISSN: 0001-0782. DOI: 10.1145/2812802. URL: http://doi.acm.org/10.1145/2812802.

[36] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, S. Miki, and S. Tagawa. "The opencl programming book". In: *Fixstars Corporation* 63 (2010), p. 11.

[37] H. Wang, F. Zhu, B. Xiao, L. Wang, and Y.-G. Jiang. "GPU-based MapReduce for large-scale near-duplicate video retrieval". In: *Multimedia Tools and Applications* 74.23 (2015), pp. 10515–10534. ISSN: 1573-7721. DOI: 10.1007/s11042-014-2185-x. URL: https://doi.org/10.1007/s11042-014-2185-x.

[38] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. "Gunrock: A High-Performance Graph Processing Library on the GPU". In: *CoRR* abs/1501.05387 (2015). arXiv: 1501.05387. URL: http://arxiv.org/abs/1501.05387.

[39] Yahoo! URL: https://www.flickr.com/.